

Passivity and Independence*

Uday S. Reddy

University of Illinois at Urbana-Champaign

Email: reddy@cs.uiuc.edu

Abstract

Most programming languages have certain phrases (like expressions) which only read information from the state and certain others (like commands) which write information to the state. These are called *passive* and *active* phrases respectively. Semantic models which make these distinctions have been hard to find. For instance, most semantic models have expression denotations that (temporarily) change the state. Common reasoning principles, such as the Hoare's assignment axiom, are not valid in such models. We define here a semantic model which captures the notions of "change", "absence of change" and "independent change" etc. This is done by extending the author's "linear logic model of state" with dependence/independence relations so that sequential traces give way to pomset traces.

1 Introduction

The problem of readers and writers [5, 10] has fascinated computer scientists for a long time. Consider a shared resource (e.g., a disk drive) that is used by concurrent client processes. Two processes that only *read* information can be allowed concurrent access to the resource. But, two writers, or a reader and writer must not access the resource concurrently to avoid interference.

The same kind of issue arises in programming languages and their semantics. Even in languages that are traditionally considered "sequential", a procedure and its arguments are best thought of as being executed concurrently, because their access to state exhibits a complex interleaved pattern. Concurrency in this setting, is a semantic abstraction. Phrases like expressions are readers of state information and phrases like commands are writers. Their access to state must be controlled by similar rules and this needs to be exhibited in the semantics.

While many languages paid attention to interference control issues, e.g., [4], Reynolds seems to have been the first to carry out a substantial study of these issues in a series of papers devoted to Algol and its programming logic [25, 26, 27, 28]. In his terminology, phrases that only read information from the state are *passive* and others *active*. Passive phrases can have concurrent access to program variables while active phrases must not. While these concepts seem fairly clear at the language level as well as in the operational semantics, denotational models that capture these principles have been hard to find.

To appreciate the difficulty, consider the type **comm** \rightarrow **exp** denoting functions from commands to expressions. Since expressions are readers and commands are writers, one cannot use a writer in any nontrivial fashion and yet produce a reader. Thus, the only functions of the **comm** \rightarrow **exp** must be constant functions. However, this is not the case if we treat **comm** as [state \rightarrow state] and **exp** as [state \rightarrow val]. The function

$$f : [\text{state} \rightarrow \text{state}] \rightarrow [\text{state} \rightarrow \text{val}] \\ f(c)(\sigma) = c(\sigma)(\alpha_0)$$

(run the input command once, then read a particular location α_0 and discard the changes to state) is a possible denotation of type **comm** \rightarrow **exp**. It is evidently not a constant function.

The problem is that states are *historical* entities (extended in time) whereas the traditional denotational models treat states as ordinary static values. To capture the property that passive computations do not change the state, one must first have a notion of *change*. This is not to be found in the traditional models of programming languages.

In previous work [23, 24], we have developed a semantic framework that models "historicity" (also called "single-threadedness") in imperative programming computations. This sets the stage for studying the more sophisticated issues like passivity and independence. However, the coherent space-based model presented in [24] turns out to be inadequate. We need

*To appear in LICS 94.

a fundamental extension for modelling independence (the absence of interference). Once this is done, passivity is found to follow.

In [25, 28], Reynolds states the following axiom for passive values: “Passive phrases, which perform no assignment or other actions that could cause interference, do not interfere with one another.” In other words, passive phrases are independent of all passive phrases, including themselves. It turns out that we can essentially take this to be the definition of passivity. Thus, once we have a model of independence, passivity can be interpreted as “full independence”.

In this paper, we present a semantic model of higher-order interference-controlled programming languages which fully accounts for historicity and passivity. Every type θ has a passive subtype θ' which includes all the passive information of θ . A function $f : \theta \rightarrow \phi$ to a passive type ϕ is then equivalent to a function $f' : \theta' \rightarrow \phi$. For example, $\mathbf{comm} \rightarrow \mathbf{exp}$ mentioned above is equivalent to $\mathbf{ns} \rightarrow \mathbf{exp}$ where \mathbf{ns} , denoting the type with a unique undefined value, is the passive subtype of \mathbf{comm} . Thus, all functions of type $\mathbf{comm} \rightarrow \mathbf{exp}$ are constant functions.

This is the first known semantic model of imperative programming which accounts for passivity in this fashion. On the other hand, as in [23, 24], our model will have an “intensional” flavor in that intermediate states will be represented in the semantics. This is mainly for pedagogical reasons. The intensional flavor gives a better focus on the issues of historicity, change, and absence of change.

Related work There is a long tradition to the semantics of imperative programs. The reader would be familiar with the Scott-Strachey approach of modelling commands as state-to-state functions [31]. This semantics has been criticized as being insufficiently abstract in dealing with *locality* of variables, and a number of improvements have been proposed based on the “possible worlds” model [26, 21, 13, 32, 19]. The recent work [20] seems to be the most advanced contribution along this line. None of these models account for *historicity* of dynamic objects (closely related to notions of “single-threadedness” and “object identity”). See discussion at the end of [20]. The previous work of the author [23, 24], based on linear logic concepts, was the first to account for historicity together with a considerably simpler model of local variables. (It should be pointed out that much work on concurrency, e.g., [11, 14], accounts for historicity, but in an untyped, nondeterministic, first-order setting.)

Syntactic control of interference has been studied

by O’Hearn [17, 18] in the possible-world framework. His work uncovered fundamental connections with linear logic and provided much inspiration to the author. The reader will find many similarities of the present model with O’Hearn’s, but she would also note that O’Hearn’s model fails the passivity criterion mentioned above. The essential problem again is that his model does not have a notion of change.

Among all the “possible-world” models, Tennent’s model of specification logic [32] stands alone in giving an accurate account of passivity. After seeing the present work, O’Hearn and Tennent have extended this model to syntactic control of interference. However, challenges remain in accounting for locality as well as historicity in this setting.

Girard’s linear logic [6] has been a source of many new ideas. The knowledgeable reader will find these ideas throughout the present work. Wadler [33] considered the issue of modelling (limited forms of) state-manipulation via linear functional programming, and noted that “read-only types” (our passive types) could not be easily obtained in this framework. Our move from coherent spaces (the semantic home of linear logic) to dependence spaces (Sec. 3) reflects a similar sentiment.

The semantic models we develop continue to show resemblances to models of concurrency. While the earlier work showed connections to models of interleaved concurrency, the present work is closely related to “parallel” or “pomset” models of concurrency [12, 22]. **Overview** After reviewing the background of Algol-like languages and the linear logic model of state in Sec. 2, we give a high-level sketch of the semantic model in Sec. 3. Sec. 4 contains the details of the model *per se*. Sec. 5 gives a semantic interpretation of interference-controlled Algol.

2 Background

2.1 Algol-like languages

We use the framework of Algol-like languages (based on Algol 60 [15]) for presenting the semantic ideas. Algol, in Reynolds’s formulation [26], is a typed lambda calculus with the following primitive types:¹

- δ **var**: the type of variables (storage cells) holding δ -typed data values,
- δ **exp**: the type of expressions (state readers) yielding δ -typed data values, and

¹Since this terminology conflicts with the usual lambda calculus terminology, the term “identifier” is used for variables in the sense of lambda calculus and the term “phrase” is used for terms/expressions.

$0, 1, \dots$	$:$	exp
$+, -, \dots$	$:$	exp \times exp \rightarrow exp
skip	$:$	comm
$_;$	$_$	comm \times comm \rightarrow comm
if0	$:$	exp \times $\theta \times \theta \rightarrow \theta$
rec	$:$	$(\theta \rightarrow \theta) \rightarrow \theta$
deref	$:$	var \rightarrow exp
$_ := _$	$:$	var \times exp \rightarrow comm
new	$:$	$(\mathbf{var} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$

Table 1: Constants in Algol-like languages

- **comm**: the type of commands (state writers).

For simplicity of presentation, we assume a single data type **int** and abbreviate **int var** (and **int exp**) to **var** (and **exp**). The syntax of types is then

$$\theta ::= \mathbf{var} \mid \mathbf{exp} \mid \mathbf{comm} \mid \theta_1 \times \theta_2 \mid \theta_1 \rightarrow \theta_2$$

Table 1 shows sample constants one finds in Algol-like languages.

2.2 Interference control

Syntactic Control of Interference, also due to Reynolds [25, 28], is a type system for avoiding the problems of aliasing and its higher-order analogue (called “interference”). The essential idea is that in a function application MN , the phrases M and N should be independent, i.e., M should not read or write to a variable that N writes to and *vice versa*. Thus, we can think of the computations of M and N as being concurrent. There is no need to reason about the complex patterns of interleaved memory access made by the two computations. To drive this point home, Reynolds also adds an independent parallel composition construct $M \parallel N$ whose formation rules are the same as those of application.

Making the restriction that a function and its argument are independent gives the effect that all free identifiers of a phrase are independent. Thus, independence can be syntactically ensured by requiring that subphrases should not share common free identifiers. The type rules for application and parallel composition are then written as follows:

$$\frac{\Gamma \vdash M : \theta \rightarrow \theta' \quad \Delta \vdash N : \theta}{\Gamma, \Delta \vdash MN : \theta'}$$

$$\frac{\Gamma \vdash M : \mathbf{comm} \quad \Delta \vdash N : \mathbf{comm}}{\Gamma, \Delta \vdash M \parallel N : \mathbf{comm}}$$

One should, however, permit concurrent phrases to share identifiers that they only read from. To admit this, Reynolds, identifies a subclass of types called *passive* types:

$$\text{passive types } \phi ::= \mathbf{exp} \mid \phi_1 \times \phi_2 \mid \theta_1 \rightarrow_P \theta_2$$

The type constructor \rightarrow_P is a new addition. It denotes “passive functions” that only read from global identifiers. The sharing of passive free identifiers is then permitted by adding a contraction rule:

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

The constants of Table 1 remain the same except that the type of **rec** is modified to use the passive function space: **rec** : $(\theta \rightarrow_P \theta) \rightarrow \theta$.

2.3 Linear logic model of state

A program was initiated in [24] to carry out a semantic analysis of imperative programming using the ideas of linear logic [6]. The initial motivation was to find a “logical foundation” for state-manipulation facilities. While this goal is still some distance away, significant semantic insights were obtained through this study. Two particular benefits derived were (i) an especially simple model of *local variables* (free of locations and global stores) and (ii) a natural formulation of *historicity* of state-encapsulating objects.

Recall that the starting point of Girard’s linear logic was a decomposition of the usual function space $A \rightarrow B$ into two operations $!A \multimap B$ [8]. The “!” construction, often called a “storage operator”, captures the structure involved in “multiple uses” of a variable. For modelling state-encapsulating objects, it was recognized in [24] that a different storage operator “ \dagger ” was needed. While “!” allows a value to be reused in an arbitrary fashion, “ \dagger ” allows it to be sequentially reused. The structure of such sequential reuse gives rise to “historicity”.

From a programming point of view, values of a type $\dagger A$ are viewed as “objects” with internal state and externally visible operations. Carrying out one of these operations may alter the internal state.

In Fig. 1, we show the historical structure of three separate objects. The first object, called a *stepper*, has a single operation to fetch an integer value. Each use of this operation alters the internal state of the stepper. Mathematically, the object can be viewed as just the set of all sequences of observable values (called “trace sets”):

$$\langle \rangle, \langle 0 \rangle, \langle 0, 1 \rangle, \dots$$

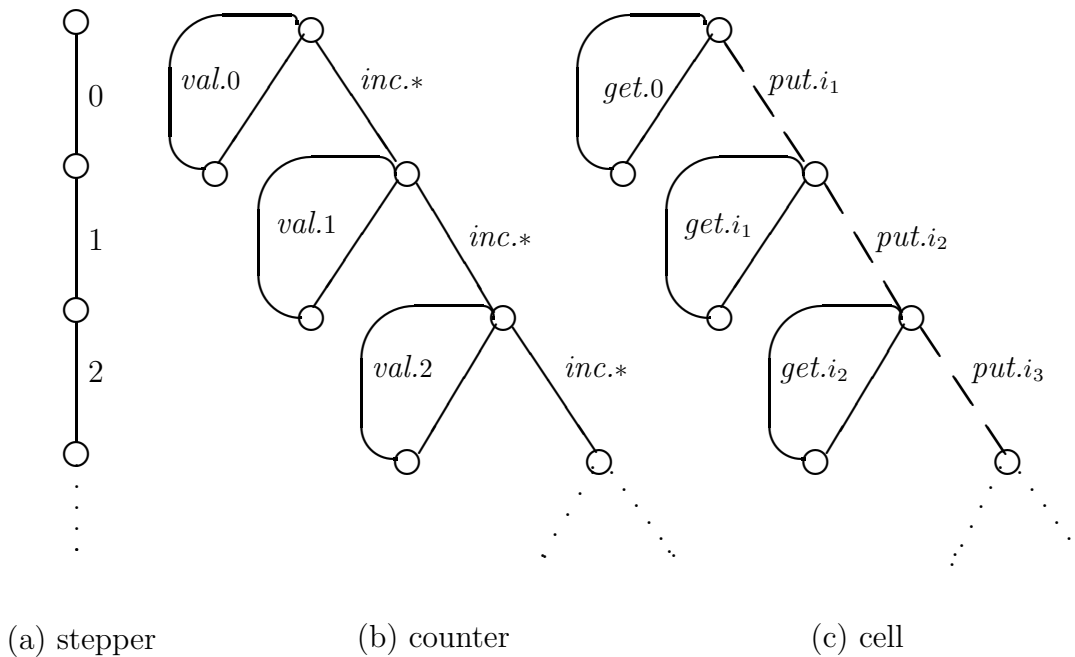


Figure 1: Example objects

The second object, called a *counter* has two operations: *val* for fetching the current value of the counter, and *inc* for incrementing it. Notice that each use of the *val* operation returns the object to essentially the same state as the original one. This is represented by the back arcs in the diagram. Operations that only read information from objects often exhibit such behavior. The third object called a *cell* has a *get* operation to fetch the current value and a *put* operation to store a new value. Variables of Algol are interpreted as objects of this kind. The reader can easily construct the trace sets for each of these objects.

While this model provides a good account of historicity, one should say that it has too much historicity. It makes no distinction between reads and writes, and, so, multiple reads are treated in a sequential manner. In an application phrase MN , the two phrases cannot share objects that they only read from. Secondly, if we construct a *composite* object by putting together two independent objects, we should be able to use the components in concurrent contexts. There was no mechanism to separate composite objects into their independent components. These are the issues of *passivity* and *independence* treated in the present work.

3 A sketch of the semantic model

It has become fairly common to treat semantic values as sets of *tokens*. A token captures the information obtained via an *observation* of a computational value and the value itself is then viewed as the set of observations that it supports. Semantic frameworks such as information systems, event structures, coherent spaces, concrete data structures and game semantics [2, 3, 9] take this kind of an approach. In the current treatment, I use the framework of coherent spaces (mainly due to their simplicity) but it is very likely that the ideas can also be adapted to some of these other frameworks.

Consider a coherent space equipped with a symmetric binary relation of *dependence*. Two tokens are deemed dependent if the observations represented by them are possibly interfering, e.g., one of them causes state changes which affect the other observation. A coherent space with such a dependence relation will be called a *dependence space*.

Linear maps can be extended to dependence spaces so that they preserve the dependence relation. This captures the notion that a function cannot produce independent outputs from dependent (possibly interfering) inputs. For example, there are no nontrivial linear maps from commands to expressions because all tokens of the latter are independent whereas no two command tokens are independent.

Dependence spaces with linear maps form a symmetric monoidal closed category.

A dependence space with an empty dependence relation has the property that all its tokens are independent of each other. So, it is called a *passive* space. It is found that every dependence space has a (universal) passive subspace satisfying the passivity criterion: every linear map from A to a passive space P uniquely factors through the passive subspace of A .

A certain form of co-algebra in independence spaces is called an *consequential space*. Values in these spaces allow *multiple* observations to be carried out in sequence, i.e., a token of a consequential space is akin to a *sequence* of tokens of a basic independence space. Each observation can potentially affect the later observations; hence, the name “consequential” space.

Some of these sequences (or some segments of such sequences) would be composed of mutually *independent* tokens. In this case, there is no required relative order for their observation. Hence, tokens of consequential spaces are really pomsets (partially ordered multisets) of tokens rather than sequences. Following [12], we call such pomsets *traces* and borrow some results from trace theory.

The co-free consequential space generated by A is denoted $\dagger A$. \dagger extends to a comonad on dependence spaces. In the standard fashion, co-algebra morphisms of such consequential spaces are representable as the arrows of the Kleisli category. Explicitly, homomorphisms $f : \dagger A \rightarrow \dagger B$ correspond to linear maps $F : \dagger A \multimap B$. We call maps of this form *Algol maps*.

Dependence spaces with Algol maps form another symmetric monoidal category. The monoidal structure in this case is a construction called *independent product* and \dagger maps independent products to tensor products: $\dagger(A \star B) \cong \dagger A \otimes \dagger B$. Notice that this is similar to the Seely isomorphism for linear logic: $!(A \& B) \cong !A \otimes !B$ [29].

The independent product is the categorical product in the subcategory passive spaces. So, we have diagonal maps $\delta : P \rightarrow P \star P$ which can be used to interpret the contraction rule for passive types.

Thus, the category of dependence spaces with Algol maps has all the structure needed to interpret interference-controlled Algol.

4 Dependence Spaces

4.1 Definition A *dependence space* is a coherent space $A = (|A|, \circlearrowleft_A)$ equipped with a symmetric

binary relation ∇_A (*dependence* or *interference*) satisfying the following axioms:

- (i) $\nabla_A \subseteq \circlearrowleft_A$, and
- (ii) $\alpha \nabla \alpha'$ implies $\alpha \nabla \alpha$ or $\alpha' \nabla \alpha'$.

The complement (*independence*) of ∇_A is denoted Δ_A . We write $\alpha \nabla_A \alpha'$ as $\alpha \nabla \alpha' [\text{mod } A]$ (or, merely, $\alpha \nabla \alpha'$ if the context makes it clear).

Dependence spaces are used to model the transitions of state machines as in Fig. 1. Intuitively, two tokens are considered coherent if they are both available as transitions from any given state. Two coherent tokens are considered dependent if the state changes affected by the two transitions (if any) may possibly interfere. Contrapositively, two independent transitions can be carried out concurrently and their total effect can be interpreted as the sum of their individual effects. Note that independence of tokens necessarily means the absence of interference while dependence of tokens only means the *possibility* of interference.

A token is considered *passive* if it is independent of itself. The axiom (ii), viewed contrapositively as $\alpha \Delta \alpha \wedge \alpha' \Delta \alpha' \implies \alpha \Delta \alpha'$, means that two passive tokens α and α' are always independent. This essentially codifies Reynolds’s axiom of passivity [25]. See also [18].

4.2 Examples The following are elementary examples of dependence spaces:

- (integer) expressions
 $exp = int$ with $i \nabla j [\text{mod } exp]$ for no i, j
- active integers
 $aint = int$ with $i \nabla j [\text{mod } aint] \iff i = j$
- commands
 $comm = \mathbf{1}$ with $* \nabla * [\text{mod } comm]$
- (integer) acceptors
 $acc = int^\perp$ with $i \nabla j [\text{mod } acc]$ for all i, j

where int is the discrete coherent space of integers (with $|int| = \omega$ and \circlearrowleft_{int} as the identity relation). Note that exp consists of only passive tokens and other spaces consist of all active tokens.

For interpreting variables, define a dependence space var as follows:

$$\begin{aligned}
|var| &= \{ \text{get}.i : i \in |int| \} \cup \{ \text{put}.j : j \in |int| \} \\
l.i \circlearrowleft l'.j [\text{mod } var] &\iff (l = l' = \text{get} \implies i = j) \\
l.i \nabla l'.j [\text{mod } var] &\iff (l = \text{put} \vee l' = \text{put}) \\
l.i \Delta l'.j [\text{mod } var] &\iff (l = l' = \text{get})
\end{aligned}$$

A token $\text{get}.i$ corresponds to the operation of a reading a value i from a variable, and a token $\text{put}.j$ corresponds to writing a value j in a variable. The dependence relation states that a put token interferes

with every token. Two get tokens, on the other hand, are independent.

4.3 Definition A dependence space is said to be *passive* if all its tokens are passive. In that case, the dependence relation is empty and, equivalently, the independence relation is full. So, a passive dependence space is essentially a coherent space. Of the above dependence spaces, exp is the only passive space. The empty dependence space \top is also passive.

4.4 Definition The (*universal*) *passive subspace* of A , denoted $\wp A$, has tokens $|\wp A| = \{\alpha \in |A| : \alpha \Delta \alpha\}$ and the corresponding restrictions of the coherence and dependence relations of A . (This is similar to what is denoted $!A$ in [17].) Evidently, $\wp A$ is passive, and $\wp P = P$ for any passive space P .

For example, $\wp(exp) = exp$, $\wp(acc) = \wp(comm) = \top$, and $\wp(var) \cong exp$.

Two dependence spaces are *equivalent*, written $A \cong B$, if there is a bijection between $|A|$ and $|B|$ that preserves coherence and dependence. (This is an isomorphism in a certain category.)

4.5 Notation We write the elements of a disjoint union $S_1 + S_2$ as $1.x$ and $2.y$, i.e.,

$$S_1 + S_2 = \{1.x : x \in S_1\} \cup \{2.y : y \in S_2\}$$

For mnemonic value, we often use the notation of “labelled” sums:

$$(a : S_1) + (b : S_2) = \{a.x : x \in S_1\} \cup \{b.y : y \in S_2\}$$

4.6 Definition There are two separate “product” constructions for dependence spaces. The *independent product* $A_1 \star A_2$ rules out any interference between the components. In contrast, the *dependent product* $A_1 \& A_2$ permits interference between the components:

$$\begin{aligned} |A_1 \star A_2| &= |A_1| + |A_2| \\ i.\alpha \circ j.\alpha' \text{ [mod } A_1 \star A_2] &\iff i = j \implies \alpha \circ \alpha' \\ i.\alpha \nabla j.\alpha' \text{ [mod } A_1 \star A_2] &\iff i = j \wedge \alpha \nabla \alpha' \\ i.\alpha \Delta j.\alpha' \text{ [mod } A_1 \star A_2] &\iff i = j \implies \alpha \Delta \alpha' \\ |A_1 \& A_2| &= |A_1| + |A_2| \\ i.\alpha \circ j.\alpha' \text{ [mod } A_1 \& A_2] &\iff \\ i = j \implies \alpha \circ \alpha' & \\ i.\alpha \nabla j.\alpha' \text{ [mod } A_1 \& A_2] &\iff \\ (i = j \implies \alpha \nabla \alpha') \wedge (\alpha \nabla \alpha \vee \alpha' \nabla \alpha') & \\ i.\alpha \Delta j.\alpha' \text{ [mod } A_1 \& A_2] &\iff \\ (i = j \wedge \alpha \Delta \alpha') \vee (\alpha \Delta \alpha \wedge \alpha' \Delta \alpha') & \end{aligned}$$

The independent product takes tokens of A_1 and A_2 to be independent of each other whereas the dependent product takes them to be interfering.

The independent product is useful for forming data structures with independent components, such as Pascal records. The dependent product is useful for

forming “objects” with multiple operations on shared state. For example,

$$\begin{aligned} var &= (\text{get} : exp) \& (\text{put} : acc) \\ counter &= (\text{val} : exp) \& (\text{inc} : comm) \end{aligned}$$

where var is as defined in 4.2 and $counter$ models the transitions of the counter object (Fig. 1).

4.7 Lemma $\wp(A \star B) = \wp A \star \wp B = \wp A \& \wp B = \wp(A \& B)$.

So, \star and $\&$ coincide for passive spaces.

4.1 Traces

The first application of dependence/independence relations is to define traces with intrinsic concurrency.

4.8 Definition A *partially ordered multiset* (pomset, for short) is a pair $s = (X_s, <_s)$ of a multiset and a partial order [22]. We often write $\alpha \in s$ to mean $\alpha \in X_s$.

We find it convenient to treat multisets as sets of labelled values α^l where l belongs to a countable set of labels. Whenever we use two multisets in the same context, we assume the multisets to be relabelled apart from each other. We ignore to mention the labels unless warranted by precision.

4.9 Definition A *trace* over a dependence space A is a pomset $s = (X_s, <_s)$ where X_s is a multiset over $|A|$ and $<_s$ is a *least* partial order on X_s satisfying

$$\alpha, \beta \in X_s \wedge \alpha \nabla \beta \implies \alpha <_s \beta \vee \beta <_s \alpha$$

The qualification “least” means that whenever $\alpha <_s \beta$, there exists $\beta' \in X_s$ such that $\alpha <_s \beta' \leq_s \beta$ and $\alpha \nabla \beta'$. All incomparable elements of the pomset are independent, i.e., all dependent pairs of elements are ordered one way or another.

A trace always has a “successor” relation $\rightarrow_s = <_s \cap \nabla_A$. Note that $<_s$ is the transitive closure of \rightarrow_s . The pair (X_s, \rightarrow_s) , often called the “dependence graph” of s , is also used to denote the trace s .

4.10 Example Consider the Algol phrase:

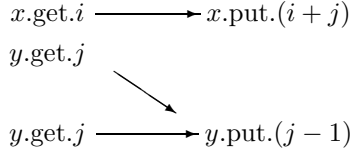
$$x : \mathbf{var}, y : \mathbf{var} \vdash (x := x + y; y := y - 1) : \mathbf{comm}$$

The “store” in this case is a composite object consisting of two variables. An execution of the command might perform a sequence of transitions on the store object of the following form:

$$\langle x.\text{get}.i, y.\text{get}.j, x.\text{put}.(i + j), y.\text{get}.j, y.\text{put}.(j - 1) \rangle$$

Each element of the sequence is a token of the dependent space $(x : var) \star (y : var)$. The sequence evidently overspecifies the temporal restrictions. The

minimal restrictions are represented by a trace with the following dependence graph:



Note that the two get's on y are independent and all the operations on x are independent from those on y .

This example also suggests that one can use sequences to compactly represent traces:

4.11 Definition Let A be a dependence space. A sequence of tokens $\langle \alpha_1, \dots, \alpha_n \rangle$ denotes a trace s with

$$\begin{array}{ccc}
X_s & = & \{\alpha_1, \dots, \alpha_n\} \\
\alpha_i \rightarrow_s \alpha_j & \iff & i < j \wedge \alpha_i \nabla_A \alpha_j
\end{array}$$

4.12 Traces over a dependence space A have an obvious monoid structure. The empty pomset \emptyset is the unit and multiplication is:

$$s \cdot t = (X_s \cup X_t, \rightarrow_s \cup \rightarrow_t \cup ((X_s \times X_t) \cap \nabla_A))$$

Two traces are *independent*, $s \Delta t$, iff $\forall \alpha \in s, \beta \in t, (\alpha \Delta \beta)$. Multiplication of independent traces is commutative:

$$s \Delta t \implies s \cdot t = t \cdot s$$

In this case, we also write $s \cdot t$ as $s \cup t$.

4.13 Definition If A is a dependence space, we define a space “repetitive A ” ($\dagger A$) for modelling objects with A type transitions.

A trace s over A is said to be *coherent* if every incomparable pair of elements in s is coherent, i.e.,

$$\forall \alpha, \beta \in s, \alpha <_s \beta \vee \beta <_s \alpha \vee \alpha \circ \beta \text{ [mod } A]$$

The *predecessor trace* of $\alpha \in s$ is a trace $\downarrow_\alpha s = \{\beta \in X_s : \beta <_s \alpha \wedge \beta \nabla \beta \text{ [mod } A]\}$ with the corresponding restriction of $<_s$.

The space $\dagger A$ is defined as:

$$\begin{array}{l}
|\dagger A| = \text{the set of coherent traces over } A \\
s \circ t \text{ [mod } \dagger A] \iff \forall \alpha \in s, \beta \in t, \\
\downarrow_\alpha s = \downarrow_\beta t \implies \alpha \circ \beta \\
s \nabla t \text{ [mod } \dagger A] \iff \exists \alpha \in s, \beta \in t, (\alpha \nabla \beta) \\
s \Delta t \text{ [mod } \dagger A] \iff \forall \alpha \in s, \beta \in t, (\alpha \Delta \beta)
\end{array}$$

Intuitively, two traces are coherent if they can both be observed from the same object. Note that formalizing this involves past-future causality. Whenever two events share the same “past”, they should be coherent. Independence of traces models concurrency without any synchronization.

4.14 Lemma $\wp(\dagger A) = \dagger \wp A = !\wp A$.

A coherent trace $s \in |\dagger A|$ is passive iff each $\alpha \in s$ is passive. This gives the first equality above. But, now, s is just a coherent multiset. So, \dagger coincides with $!$ for passive spaces (where the latter is the co-free co-commutative comonoid construction, as in [7]).

4.2 The structure of dependence spaces

The second application of dependence/independence relations is to define an appropriate notion of functions.

4.15 Definition A *linear map* between dependence spaces $f : A \multimap B$ is a linear map of the underlying coherent spaces which also preserves dependence. That is, for all $(\alpha, \beta), (\alpha', \beta') \in f$,

$$\bullet \alpha \nabla \alpha' \text{ [mod } A] \implies \beta \nabla \beta' \text{ [mod } B].$$

This condition states that a function cannot “manufacture” independence. If the inputs are interfering, the outputs will be interfering too. In other words, interference is conserved. Note that $(\alpha, \beta) \in f$ only if $\alpha \nabla \alpha \implies \beta \nabla \beta$. That is, passive outputs can only arise from passive inputs.

a linear map of the form $f : \dagger A \multimap B$ is called an *Algol map* and denoted $f : A \rightarrow B$. Such maps denote functions that use their arguments multiple times.

4.16 Examples

- (i) Every linear map $f : \text{int} \multimap \text{int}$ extends to a linear map $f : \text{exp} \multimap \text{exp}$.
- (ii) There is a single linear map $f : \text{comm} \multimap \text{exp}$, viz., the empty map. This illustrates the role played by the “conservation of interference” principle. Since comm is active and exp is passive, there can be no nontrivial linear maps f of this form. Similarly, all linear maps $f : \dagger \text{comm} \multimap \text{exp}$ are constant maps.
- (iii) $\text{twice} : \text{comm} \rightarrow \text{comm} = \{(\langle *, * \rangle, *)\}$ is a linear map. This represents a procedure that runs its input command twice. In fact, all Algol maps $\text{comm} \rightarrow \text{comm}$ correspond to integers in this fashion. The function corresponding to n runs its input command n times.
- (iv) The projections

$$\begin{array}{l}
\text{readonly: } \text{var} \multimap \text{exp} = \{(\text{get}.i, i) : i \in |\text{int}|\} \\
\text{writeonly: } \text{var} \multimap \text{acc} = \{(\text{put}.i, i) : i \in |\text{int}|\}
\end{array}$$

are linear maps. More generally, there are projections for any dependent product: $\pi_i : A_1 \& A_2 \multimap A_i = \{(i.\alpha, \alpha) : \alpha \in |A_i|\}$. Similar projections are also available for $A_1 \star A_2$.

read : $\dagger A \multimap A$	$= \{ (\langle \alpha \rangle, \alpha) : \alpha \in A \}$
Dup : $\dagger A \multimap \dagger \dagger A$	$= \{ (s_1 \cdots s_n, \langle s_1, \dots, s_n \rangle) : s_1, \dots, s_n \in \dagger A \wedge s_1 \cdots s_n \in \dagger \dagger A \}$
kill : $\dagger A \multimap \mathbf{1}$	$= \{ (\emptyset, *) \}$
dup : $\dagger P \multimap \dagger P \otimes \dagger P$	$= \{ (s \cup t, (s, t)) : s, t, s \cup t \in \dagger P \}$
unpack : $\dagger(A_1 \star A_2) \multimap \dagger A_1 \otimes \dagger A_2$	$= \{ (s, (s \downarrow 1, s \downarrow 2)) : s \in \dagger(A_1 \star A_2) \}$
apply : $(A \multimap B) \otimes A \multimap B$	$= \{ (\langle (\alpha, \beta), \alpha \rangle, \beta) : \alpha \in A \wedge \beta \in B \}$
$\Lambda(f)$: $C \multimap (A \multimap B)$	$= \{ (\gamma, (\alpha, \beta)) : ((\gamma, \alpha), \beta) \in f \}$
apply _A : $(A \rightarrow B) \star A \rightarrow B$	$= \{ (\langle (1.(s, \beta)) \cup (\{2\} \times s), \beta \rangle) : s \in \dagger A \wedge \beta \in B \}$
$\Lambda_A(f)$: $C \rightarrow (A \rightarrow B)$	$= \{ (s \downarrow 1, (s \downarrow 2, \beta)) : (s, \beta) \in f \}$

Table 2: Maps involved in the structure

- (v) If $f : B \multimap A_1$ and $g : B \multimap A_2$ are linear maps, then $\langle f, g \rangle : B \multimap A_1 \& A_2 = \{ (\beta, 1.\alpha) : (\beta, \alpha) \in f \} \cup \{ (\beta, 2.\alpha) : (\beta, \alpha) \in g \}$ is a linear map.
- (vi) We can make a counter from a variable using the Algol map $c : \text{var} \rightarrow \text{counter}$. Explicitly, c is

$$\begin{aligned} & \{ (\langle \text{get}.i, \text{val}.i \rangle : i \in |\text{int}|) \cup \\ & \{ (\langle \text{get}.i, \text{put}.(i+1) \rangle, \text{inc}.*) : i \in |\text{int}| \} \end{aligned}$$

This map corresponds to a phrase of the form

$$x : \text{var} \vdash (x, x := x + 1) : \text{exp} \times \text{comm}$$

4.17 Theorem

- (i) *Dependence spaces with linear maps form a cartesian category* DEPL with binary products $\&$ and terminal object \top .
- (ii) *Passive dependence spaces form a full subcategory* PASL of DEPL.
- (iii) PASL is equivalent to COHL.

There exists a linear injection $\mathbf{act} : \wp A \multimap A$. More interestingly, there is a map $\mathbf{pas} : A \multimap \wp A = \{ (\alpha, \alpha) : \alpha \in |\wp A| \}$ which is a universal arrow in the following sense.

4.18 Lemma If $f : A \multimap P$ is a linear map, where P is passive, then there exists a unique linear map $f' : \wp A \multimap P$ such that $f = \mathbf{pas}; f'$.

This crucial property makes it obvious that the function space $\text{comm} \multimap \text{exp}$ is trivial. In categorical terminology, the property means the following:

4.19 Theorem PASL is a reflective subcategory of DEPL.

The \wp operator may be viewed as a functor $\wp : \text{DEPL} \rightarrow \text{PASL}$. This is left adjoint to the inclusion functor $\mathcal{I} : \text{PASL} \rightarrow \text{DEPL}$ with the unit of the adjunction $\mathbf{pas} : \mathbf{I} \rightarrow \mathcal{I}\wp$. Note that this makes $\mathcal{I}\wp$ into a monad with $(\mathcal{I}\wp)^2 = \mathcal{I}\wp$. It turns out that $\mathcal{I}\wp$ is also a comonad with \mathbf{act} as the counit (Cf. [18]), but it is the monad structure that seems more useful.

4.20 Lemma

- (i) \dagger extends to a comonad in DEPL.
- (ii) $\dagger P$ forms a co-commutative comonoid in the monoidal structure $(\text{PASL}, \otimes, \mathbf{1})$.
- (iii) There are isomorphisms $\dagger \top \cong \mathbf{1}$ and $\dagger(A_1 \star A_2) \cong \dagger A_1 \otimes \dagger A_2$.

The requisite linear maps for these statements may be found in Table 2. The tensor products are a natural generalization from coherent spaces. Use the dependence relations:

$$\begin{aligned} (\alpha, \beta) \nabla (\alpha', \beta') \ [\text{mod } A \otimes B] & \iff \\ (\alpha \nabla \alpha' \wedge \beta \supset \beta') \vee (\alpha \supset \alpha' \wedge \beta \nabla \beta') & \\ * \nabla * \ [\text{mod } \mathbf{1}] & \iff \text{false} \end{aligned}$$

4.21 Definition The linear function space of dependence spaces is the following:

$$\begin{aligned} |A \multimap B| & = \{ (\alpha, \beta) \in |A| \times |B| : \alpha \nabla \alpha \implies \beta \nabla \beta \} \\ (\alpha, \beta) \supset (\alpha', \beta') \ [\text{mod } A \multimap B] & \iff \\ \alpha \supset \alpha' \implies \beta \supset \beta' \wedge \alpha \frown \alpha' \implies \beta \frown \beta' \wedge & \\ \alpha \nabla \alpha' \implies \beta \nabla \beta' & \\ (\alpha, \beta) \nabla (\alpha', \beta') \ [\text{mod } A \multimap B] & \iff \\ \alpha \supset \alpha' \implies \beta \nabla \beta' & \end{aligned}$$

Note that a linear map $f : A \multimap B$ is nothing but a coherent set of this function space.

It is easy to verify that \multimap is an internal hom of DEPL. We have a natural isomorphism

$$\text{DEPL}(C \otimes A, B) \cong \text{DEPL}(C, A \multimap B)$$

with the combinators \mathbf{apply} and Λ shown in Table 2.

4.22 Theorem $(\text{DEPL}, \otimes, \mathbf{1}, \multimap)$ is symmetric monoidal closed.

4.3 Structure for Algol

4.23 Theorem

- (i) *Dependence spaces with Algol maps form a cartesian category* DEPA with binary products $\&$ and terminal object \top .

- (ii) *Passive spaces form a full reflective subcategory PASA of DEPA.*
- (iii) *PASA is equivalent to COHS (the category of coherent spaces with stable maps).*

DEPA is nothing but the Kleisli category of DEPL under the comonad \dagger . So, all the requisite structure follows from Lemma 4.20. The identity map is **read** : $A \rightarrow A$. The composition $f;g$ in DEPA is the linear map **Dup**; $\dagger f;g$. The universal arrow to passive subspaces is **read**; **pas** : $A \rightarrow \wp A$.

4.24 The independent product \star plays the role of a tensor product in DEPA. (Note that it is not the categorical product.) We can define the function space $A \rightarrow B$ as the dependence space $\dagger A \multimap B$. The isomorphism

$$\text{DEPA}(C \star A, B) \cong \text{DEPA}(C, A \rightarrow B)$$

follows from Lemma 4.20(iii). The combinators **apply**_A and Λ_A are explicitly given in Table 2.

4.25 Theorem

- (i) *(DEPA, \star, \top, \rightarrow) is symmetric monoidal closed.*
- (ii) *(PASA, \star, \top, \rightarrow) is cartesian closed.*

The latter is a consequence of the fact that \star and $\&$ coincide for passive spaces.

4.26 Definition The *passive function space* $A \rightarrow_p B$ is defined the same way as $A \rightarrow B$ except that the dependence relation is empty. (Thus, $A \rightarrow_p B$ is a passive space.)

It is easy to verify the isomorphism

$$\text{DEPA}(P \star A, B) \cong \text{DEPA}(P, A \rightarrow_p B)$$

by noting that $\Lambda_A(f)$ is a valid Algol map.

5 Semantics

Given the structure of Section 4, it is straightforward to give a semantic interpretation of syntactic control of interference. The interpretation of types as dependence spaces is immediate:

$$\begin{aligned} \mathbf{exp}^\circ &= \mathit{exp} & \mathbf{comm}^\circ &= \mathit{comm} & \mathbf{var}^\circ &= \mathit{var} \\ (\theta_1 \times \theta_2)^\circ &= \theta_1^\circ \& \theta_2^\circ \\ (\theta_1 \rightarrow \theta_2)^\circ &= \theta_1^\circ \rightarrow \theta_2^\circ & (\theta_1 \rightarrow_P \theta_2)^\circ &= \theta_1^\circ \rightarrow_P \theta_2^\circ \end{aligned}$$

Note that the interpretations of passive types are passive spaces.

A phrase $x_1 : \theta_1, \dots, x_n : \theta_n \vdash p : \theta$ is interpreted as an Algol map of type $\theta_1^\circ \star \dots \star \theta_n^\circ \rightarrow \theta^\circ$. Theorem 4.25 and the other properties mentioned in Sec. 4 give the requisite structure for this interpretation. For example, function application is interpreted by

$$\llbracket MN \rrbracket = \Gamma^\circ \star \Delta^\circ \xrightarrow{\llbracket M \rrbracket \star \llbracket N \rrbracket} (\theta^\circ \rightarrow \theta'^\circ) \star \theta^\circ \xrightarrow{\mathbf{apply}_A} \theta'^\circ$$

The interpretation of the constants is shown in Table 3.

The adequacy of the semantics can be shown using techniques similar to those in [24]:

5.1 Theorem (Computational adequacy) *If $\vdash c : \mathbf{comm}$ is a command, $(\emptyset, *) \in \llbracket \vdash c : \mathbf{comm} \rrbracket$ iff $(c, \sigma_0) \Downarrow \sigma_0$ (where σ_0 is the empty state.)*

Turning attention to the issue of full abstraction, it is easy to see that the present model is not going to be fully abstract because interference-controlled Algol has PCF as its sublanguage and a coherent space-based model is not fully abstract for PCF [3]. On the other hand, our interest is really in modelling aspects of state and one must ask how well these aspects are modelled. We follow the pioneering approach of [13] and show the efficacy of the model on a selection of well-chosen test equivalences.

5.2 Example The first example of [13] tests the equivalence of the following functions of type $\mathbf{comm} \rightarrow \mathbf{comm}$:

$$\lambda P. \mathbf{new} \lambda x. P \quad \equiv \quad \lambda P. P$$

The point is that the nonlocal command (“procedure”) P must not have access to the local variable x .

It is straightforward to check the equivalence in the model. Assuming P terminates, the meaning of $\lambda x. P$ is the map $\{(\emptyset, *)\}$, and, using the interpretation of **new**, we find that **new** $\lambda x. P$ terminates. Thus, both the sides of the equivalence denote the identity map $\{((*), *)\}$.

Other examples of [13] follow similarly.

5.3 Example The last example of [13] (which was left unsolved by their semantics) is a variant of the following equivalence in the type $(\mathbf{comm} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$:

$$\begin{aligned} \lambda P. \mathbf{new} \lambda x. P(x := x + 1) &\equiv \\ \lambda P. \mathbf{new} \lambda x. P(x := x - 1) & \end{aligned}$$

It is important to note that every Algol map in $\mathit{comm} \rightarrow \mathit{comm}$ is either empty or of the form $[n] = \{((*)^n, *)\}$. The map $[n]$ corresponds to a procedure that runs its input command n times. Using the interpretation of **new**, it is again easy to verify that both the sides have the meaning

$$\{([0], *), ([1], *), \dots\}$$

The essential point is that the meaning of P has nothing to do with the internal effects of its argument.

5.4 Example A more interesting variant of the above example appears in [20]. Consider the following equivalence in the type $(\mathbf{exp} \times \mathbf{comm} \rightarrow \mathbf{comm}) \rightarrow$

$\llbracket 0 \rrbracket$	$: \top \rightarrow \text{exp}$	$= \{(\emptyset, 0)\}$
$\llbracket + \rrbracket$	$: \text{exp} \& \text{exp} \rightarrow \text{exp}$	$= \{(\langle 1.i, 2.j \rangle, i + j) : i, j \in \text{int} \}$
$\llbracket \text{diverge} \rrbracket$	$: \top \rightarrow \text{comm}$	$= \{ \}$
$\llbracket \text{skip} \rrbracket$	$: \top \rightarrow \text{comm}$	$= \{(\emptyset, *)\}$
$\llbracket ; \rrbracket$	$: \text{comm} \& \text{comm} \rightarrow \text{comm}$	$= \{(\langle 1.*, 2.* \rangle, *)\}$
$\llbracket \text{if0} \rrbracket$	$: \text{exp} \& A \& A \rightarrow A$	$= \{(\langle 1.0, 2.\alpha \rangle, \alpha) : \alpha \in A \} \cup$ $\{(\langle 2.i, 3.\alpha \rangle, \alpha) : i \in \text{int} \wedge i \neq 0 \wedge \alpha \in A \}$
$\llbracket \text{while0} \rrbracket$	$: \text{exp} \& \text{comm} \rightarrow \text{comm}$	$= \{(\langle 1.0, 2.* \rangle^n \cdot \langle 1.i, * \rangle, *) : n \geq 0 \wedge i \in \text{int} \wedge i \neq 0 \}$
$\llbracket \text{rec} \rrbracket$	$: (A \rightarrow_p A) \rightarrow A$	$= \text{fix}_A$
$\llbracket \text{deref} \rrbracket$	$: \text{var} \rightarrow \text{exp}$	$= \{(\langle \text{get}.i \rangle, i) : i \in \text{int} \}$
$\llbracket := \rrbracket$	$: \text{var} \& \text{exp} \rightarrow \text{comm}$	$= \{(\langle 2.i, 1.\text{put}.i \rangle, *) : i \in \text{int} \}$
$\llbracket \text{new} \rrbracket$	$: (\text{var} \rightarrow \text{comm}) \rightarrow \text{comm}$	$= \{(\langle (s, *) \rangle, *) : s \in \text{cell} \}$

($\text{cell} \subseteq |\dagger \text{var}|$ is the trace set of the cell object shown in Fig. 1(c).)

Table 3: Constants in Algol-like languages

comm:

$$\lambda P. \mathbf{new} \lambda x. x := 0; P(x, x := x + 1) \equiv \lambda P. \mathbf{new} \lambda x. x := 0; P(-x, x := x - 1)$$

The procedure P is being passed two different representations of a counter object. In spite of this difference, the denotation of the two objects is precisely the same, viz., the counter trace set in the dependence space $\dagger(\text{exp} \& \text{comm})$ as depicted in Fig. 1(b). The interpretation of **new** again ensures that both the sides of the equivalence have the same meaning.

These examples suggest that the *locality* of variables is modelled accurately. It should be pointed out that the recent “parametricity” models [20, 30] do so as well.

5.5 Example Consider the following equivalence in type **exp** with two free identifiers $f : \mathbf{exp} \rightarrow \mathbf{exp}$ and $x : \mathbf{var}$:

$$\mathbf{if0}(x, f(x), 1) \equiv \mathbf{if0}(x, f(0), 1)$$

If we factor the effect of multiple reads of x , the two expressions have essentially the same interpretation.

Most published models of Algol-like languages fail to satisfy this equivalence as they allow temporary side effects in expressions like $f(x)$. (Cf. Introduction.) In other words, they do not model *passivity*. The only exception is Tennent’s model of specification logic [32].

5.6 Example The following test equivalence, due to P. W. O’Hearn, illustrates *historicity* (or single-threading) of state. Assume a free identifier $P : \mathbf{comm} \rightarrow \mathbf{comm}$.

$$\mathbf{new} \lambda x. x := 0; P(x := x + 1); \mathbf{if0}(x, \mathbf{skip}, \mathbf{diverge}) \equiv P(\mathbf{diverge})$$

Informally, if P runs its argument at all, both the sides diverge. If P ignores its argument, both the sides are equivalent to $P(\mathbf{skip})$. Since $\text{comm} \rightarrow \text{comm}$ is isomorphic to flat naturals, this argument can be made formal in our semantics. Reason by cases, considering $\llbracket P \rrbracket = [0]$ and $\llbracket P \rrbracket \neq [0]$. (Cf. 5.3.)

No other known model of Algol-like languages models historicity in this fashion. See discussion in [20, Sec. 6].

While the basic ideas of the present model seem right, some of the technical details fall short. The space $\dagger \text{var}$ consists of “non-state-like” traces, e.g., $\langle \text{put}.0, \text{get}.1 \rangle$. So, $\text{var} \rightarrow \text{comm}$ has too many maps though, eventually $\llbracket \text{new} \rrbracket$ cuts them down. Moreover, traces are in any case intensional (representing intermediate states). The solution is to move to the Eilenberg-Moore category of \dagger co-algebras (“consequential spaces” of Sec. 3).

6 Conclusion

We have defined here what seems to be the first semantic model for higher-order imperative programming languages which fully models historicity and passivity. The development of the semantic notions is that, first, one needs a notion of change (or historicity), then, a notion of independence, and, finally, passivity is obtained as the absence of change and full independence.

It is interesting that significant notions of concurrency theory [12, 22, 11, 14] should be useful in this formulation. See also [1] where many similar ideas are echoed.

This work forms another step in the program

initiated in [24]. A major topic that was not addressed here is the *extensionality* of state transitions. It seems that many concepts of algebraic automata theory are closely related to the required extensional model. These are being explored. Much work remains to be done on developing reasoning principles based on these semantic models and testing them on practical applications.

Acknowledgements It is a pleasure to thank Peter O’Hearn who has been a constant source of inspiration and encouragement. His remarkable examples helped me focus on important issues as well as to improve the presentation of this work. Bob Tennent, Samson Abramsky, Radha Jagadeesan and Christian Retoré have made valuable suggestions through various discussions. This work was supported by NSF grant NSF-CCR-93-03043.

References

- [1] ABRAMSKY, S. Interaction categories and communicating sequential processes. In *A Classical Mind: Essays in Honor of C. A. R. Hoare*, A. W. Roscoe, Ed. Prentice-Hall International, 1994. To appear.
- [2] ABRAMSKY, S., AND JAGADEESAN, R. Games and full completeness for multiplicative linear logic. Electronic manuscript (ftp), theory.doc.ic.ac.uk, directory /theory/papers/Abramsky., 1992.
- [3] BERRY, G., CURIEN, P., AND LEVY, J. J. Full abstraction for sequential languages: The state of the art. In Nivat and Reynolds [16].
- [4] BRINCH HANSEN, P. *Operating Systems Principles*. Prentice Hall, 1973.
- [5] COURTOIS, P., HEYMANS, F., AND PARNAS, D. Concurrent control with readers and writers. *Comm. ACM* 14, 10 (Oct 1971), 667–668.
- [6] GIRARD, J.-Y. Linear logic. *Theoretical Comput. Sci.* 50 (1987), 1–102.
- [7] GIRARD, J.-Y. A new constructive logic: Classical logic. Prepublication 24, Equipe de Logique, University of Paris VII, May 1991.
- [8] GIRARD, J.-Y., LAFONT, Y., AND TAYLOR, P. *Proofs and Types*. Cambridge Univ. Press, 1989.
- [9] GUNTER, C. A. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
- [10] HOARE, C. A. R. Monitors: An operating system structuring concept. *Comm. ACM* 17, 10 (Oct 1974), 549–558.
- [11] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall International, London, 1985.
- [12] MAZURKIEWICZ, A. Basic notions of trace theory. In *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, vol. 354 of LNCS. Springer-Verlag, 1989, pp. 285–363.
- [13] MEYER, A. R., AND SIEBER, K. Towards fully abstract semantics for local variables. In *Fifteenth Ann. ACM Symp. on Princ. of Program. Lang.* (1988), ACM, pp. 191–203.
- [14] MILNER, R. *Communication and Concurrency*. Prentice Hall, 1989.
- [15] NAUR, P. Report on the algorithmic language ALGOL 60. *Comm. ACM* 3, 5 (May 1960), 299–314.
- [16] NIVAT, M., AND REYNOLDS, J. C., Eds. *Algebraic Methods in Semantics*. Cambridge Univ. Press, 1985.
- [17] O’HEARN, P. W. Linear logic and interference control. In *Category Theory and Computer Science*, vol. 350 of LNCS. Springer-Verlag, 1991, pp. 74–93.
- [18] O’HEARN, P. W. A model for syntactic control of interference. *Math. Struct. Comput. Sci.* 3 (1993), 435–465.
- [19] O’HEARN, P. W., AND TENNENT, R. D. Semantics of local variables. In *Applications of Categories in Computer Science*, M. P. Fourman, P. T. Johnstone, and A. M. Pitts, Eds. Cambridge Univ. Press, 1992, pp. 217–238.
- [20] O’HEARN, P. W., AND TENNENT, R. D. Parametricity and local variables. Tech. Rep. SU-CIS-93-30, Syracuse University, Oct 1993. (original version in POPL 93, pp. 171-184.).
- [21] OLES, F. J. Type algebras, functor categories and block structure. In Nivat and Reynolds [16], pp. 543–573.
- [22] PRATT, V. The pomset model of parallel processes: Unifying the temporal and the spatial. In *Seminar on Concurrency*, vol. 197 of LNCS. Springer-Verlag, 1984, pp. 180–196.
- [23] REDDY, U. S. Global state considered unnecessary: Semantics of interference-free imperative programming. In *ACM SIGPLAN Workshop on State in Program. Lang.* (June 1993), Technical Report YALEU/DCS/RR-968, pp. 120–135.
- [24] REDDY, U. S. A linear logic model of state. Electronic manuscript (ftp), cs.uiuc.edu, directory /pub/reddy/papers, Oct 1993.
- [25] REYNOLDS, J. C. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.* (1978), ACM, pp. 39–46.
- [26] REYNOLDS, J. C. The essence of Algol. In *Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds. North-Holland, 1981, pp. 345–372.
- [27] REYNOLDS, J. C. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, D. Neel, Ed. Cambridge Univ. Press, 1982, pp. 121–161.
- [28] REYNOLDS, J. C. Syntactic control of interference, Part II. In *Intern. Colloq. Aut., Lang. and Program.*, vol. 372 of LNCS. Springer-Verlag, 1989, pp. 704–722.

- [29] SEELY, R. A. G. Linear logic, *-autonomous categories and cofree coalgebras. In *Categories in Computer Science and Logic*, vol. 92 of *Contemp. Math.* AMS, 1989, pp. 371–382.
- [30] SIEBER, K. New steps towards full abstraction for local variables. In *ACM SIGPLAN Workshop on State in Program. Lang.* (June 1993), Technical Report YALEU/DCS/RR-968, Yale University, New Haven, pp. 88–100.
- [31] STOY, J. E. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* MIT Press, 1977.
- [32] TENNENT, R. D. Semantical analysis of specificaiton logic. *Inf. Comput.* 85, 2 (1990), 135–162.
- [33] WADLER, P. Is there a use for linear logic? In *Proc. Symp. on Partial Evaluation and Semantics-Based Program Manipulation* (1991), ACM, pp. 255–273. (SIGPLAN Notices, Sep. 1991).