

# Global State Considered Unnecessary: Semantics of Interference-free Imperative Programming

Uday S. Reddy

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

May 25, 1993

## 1 Introduction

Idealized Algol, due to Reynolds [Rey81], is a clean integration of functional and imperative programming features. It is a typed lambda calculus whose primitive types allow for state manipulation. These primitive types include:

- $\delta$  **var** - variables holding  $\delta$ -typed values,
- $\delta$  **exp** - state dependent expressions giving  $\delta$ -typed values, and
- **comm** - state modifying commands.

Though Algol is one of the oldest programming languages, it has had concepts that are remarkably modern. In particular, it is the only widely known imperative programming language that satisfies unrestricted  $\beta$  equivalence (often identified with “referential transparency”). Several recent proposals for integrating functional and imperative programming features [ORH93, PW93, SRI91] involve ideas resembling those of Algol, and Reynolds himself has been designing a successor to Algol called Forsythe [Rey88]. In this paper, I present a novel approach to the denotational semantics of Algol that clarifies the essential structure of state manipulation.

There is a long tradition to the semantics of imperative programs. The reader is no doubt familiar with the Scott-Strachey approach of modelling commands as state-to-state functions [Sto77]. This semantics has been criticized as being insufficiently abstract in dealing with local variables, and a number of researchers have proposed improvements [HMT83, MS88, OT92, Ole85, Rey81, Ten90]. ([OT92, Ten89] give good overviews of the extant techniques). To understand the issue of local variables consider the following equivalence:

$$\mathbf{new} \ x. \ x := 0; \ p(x := x + 1) \quad \equiv \quad p(\mathbf{skip})$$

Here, the free variable  $p$  is of type **comm**  $\rightarrow$  **comm**. The command on the left allocates a local variable  $x$ , initializes it, and calls a procedure  $p$  with an increment operation on  $x$  as the argument. (The type of  $p$  allows such an argument. In original Algol 60, such arguments were obtained by “parameterless” procedures). After the completion of the procedure call,  $x$  is deallocated. Since  $x$  is a local variable, the procedure  $p$  has no direct access to it other than that provided by its

argument. Further,  $x$  is never read before it is discarded. So, the effects of  $p(x := x + 1)$  are unobservable. Passing any command to  $p$  (such as the identity command **skip**) should be equivalent to this. However, no semantics that validates this equivalence was known until recently. In work done independently of mine, O’Hearn and Tennent [OT93] proposed a refinement of the “possible world” semantics, which handles equivalences of this form. However, it seems that their semantics is excessively complicated due to its reliance on global states.

Another problem is modelling the fact that imperative programs use the state in a “single-threaded” fashion. This is illustrated by the following equivalence:<sup>1</sup>

$$\text{if } x = 0 \text{ then } f(x) \text{ else } 1 \quad \equiv \quad \text{if } x = 0 \text{ then } f(0) \text{ else } 1$$

where  $f : \text{int exp} \rightarrow \text{int exp}$ . Clearly, the  $r$ -value of  $x$  and 0 are the same in the then-branch of the conditional. However, the semantics of [OT93] (as well as most other models in the literature) fails to validate this equivalence. Essentially, they allow the semantic function  $f$  to access the global state. So,  $f(x)$  can temporarily modify the value of  $x$  and reset it before completion (violating single-threadedness) whereas  $f(0)$  cannot do so. This (undesirable) feature of the semantics is sometimes called a “snap-back” effect.

In the summer of 1992, I initiated a fresh approach to the semantics of imperative languages based on the insights obtained from linear logic [Gir87]. (Some of these results are reported in [Red93b, Red93a]). Being a radically different approach to semantics, it stumbles on many open issues which need further resolution. In this paper, I outline the bare bones of this approach in a conventional domain-theoretic framework. Being a short study, I will limit attention to an interference-free fragment of Algol. (*Interference* is the higher-order analogue of aliasing. I will use here a system called “syntactic control of interference” due to Reynolds [Rey78] to prohibit interference). Also, I will focus mainly on an *intensional* version of the semantics which is simpler to understand. (“intensional” meaning that some of the internal structure of the programs appears in the semantics in addition to the observable behavior). The extensional version of the semantics, using coherent spaces, is briefly sketched in Section 4.

## 1.1 The problem

The traditional semantics of imperative features is based on the notion of a *global store*. A variable is an index into the store and the store maps such indices to classical *static* values. The problem with this approach is that the store becomes a conceptual bottleneck (recall the “von Neumann bottleneck” of [Bac78]). Different portions of the program work on different parts of the store, possibly extending them in separate ways, and threading a single global store through all of them becomes a heavy burden. In our approach, we will decompose the store into tiny pieces (as small as individual variables) and these pieces can be manipulated independently. But, the cost to pay for this flexibility is that we must give up the conventional static values and come to grips with a new kind of semantic values, *viz.*, *active values*. Most of our energy will be concentrated on this aspect. The semantics is relatively straightforward once active values are mastered.

## 1.2 A word about Syntactic Control of Interference

The central idea of syntactic control of interference (SCI) [Rey78, Rey89] is that, in a function application  $MN$ ,  $M$  and  $N$  should be “independent”, *i.e.*,  $M$  does not change something that  $N$  reads or writes and *vice versa*. Ensuring that applications always satisfy independence gives the

---

<sup>1</sup>I am indebted to Peter O’Hearn for this example.

benefit that all free variables are always independent. Thus, to check for the independence of  $M$  and  $N$ , we only need to check that they have no common free variables.

However, imposing that  $M$  and  $N$  never have common free variables is draconian. We could not write  $+ x x$  for example. To relax the restriction, we identify another class of values called *passive* values which never change the state. Passive values do not interfere with each other including themselves. Free variables denoting passive values can then be shared by  $M$  and  $N$ . While a variable (in the imperative programming sense) is an active value, a dereferenced version of a variable is passive. (This is related to the Wadler’s issue of read-only types [Wad90]). So,  $x$  in  $+ x x$  is passive and such an application is permitted. Note that passive values are state-*dependent*, but behave like static values in local contexts because they only read the state.

The active-passive distinction can be made cleanly at the level of types as follows:

$$\begin{array}{ll} \text{active types } \alpha & ::= \delta \mathbf{var} \mid \mathbf{comm} \mid \theta \rightarrow_A \alpha \mid \theta \times \alpha \mid \alpha \times \theta \\ \text{passive types } \phi & ::= \delta \mathbf{exp} \mid \phi_1 \rightarrow \phi_2 \mid \theta \rightarrow \alpha \mid \phi_1 \times \phi_2 \\ \text{all types } \theta & ::= \alpha \mid \phi \end{array}$$

Here, “ $\rightarrow$ ” denotes *passive functions* which don’t modify global variables directly (even though they can make modifications via their arguments) whereas “ $\rightarrow_A$ ” denotes *active functions* which may modify global variables. The reader interested in a fuller discussion of SCI is referred to [O’H91] which also draws an important correspondence with linear logic.

### 1.3 Related work

The previous work on semantics of local variables was already mentioned. [OT93] seems to be the most advanced contribution along this line. The reader would notice that the present semantics runs very much counter to these approaches. While they attempt to cut down a global store based semantics to meaningful denotations, we start by identifying the observable behaviors of individual state variables and build up to larger values.

Girard’s linear logic [Gir87] was an important source of new ideas for me. For pedagogical reasons, I will mention little of linear logic in this paper, but the informed reader can see linear logic ideas everywhere. I also owe considerable debt to O’Hearn’s work [O’H91] on relating linear values and active values (as in linear logic and SCI respectively). This work, together with Wadler’s related insights [Wad91], gets “halfway” towards the semantics reported here.

Another important piece of related work is Shapiro’s modelling of state in concurrent logic programming languages [Sha83]. (See also [ST86]). This is closely related to the classical stream-based approach to modelling state in functional programming [FW79, KM77, KL85]. However, concurrent logic programming languages fill in an important missing piece in this set-up (often called “logic variables”). Formalizing it requires considerable work, however. See [Laf87, Red93c]. Also related is Milner’s translation of imperative languages to CCS [Mil89, Chap. 8].

### 1.4 Overview

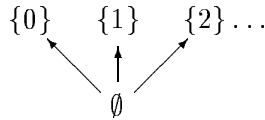
In Section 2, I briefly review the token-based approach to defining semantics and describe how dynamic variables can be modelled using histories of tokens. The discussion is informal and intuitive. In Section 3, I give an intensional semantics to interference-free Algol using these ideas. Several examples of semantics and semantic equivalences are shown. Finally, a technically precise account of everything is given Section 4 together with an extensional semantics of state manipulation.

## 2 Values and Tokens

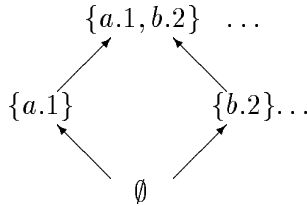
I use a token-based approach to dealing with semantic values. A *token* of some type is a single atomic piece of information about a value of that type. For example:

type	tokens
<i>bool</i>	<i>tt</i> , <i>ff</i>
<i>int</i>	0, 1, 2, ...
$A \times B$	$a.1$ where $a$ is a token of $A$ , and $b.2$ where $b$ is a token of $B$ .
$A \otimes B$	$(a, b)$ where $a$ and $b$ are tokens of $A$ and $B$ respectively.

A semantic value is a set of tokens that are “consistent” together. Such values are naturally ordered by the subset order to form a cpo. For reasons that will become clear in the sequel, I call such cpo’s *passive domains* and write them as  $Passive[A]$  (where  $A$  is a semantic type). For example, the domain  $Passive[int]$  is:



The domain  $Passive[A \times B]$  is



$A \otimes B$  is called the *tensor product*. I will not discuss it any further. I will also use a unit type  $\mathbf{1}$  which has a single token denoted  $*$ .

Tokens of this kind are in used in various semantic frameworks such as Scott’s information systems [Sco82] and Girard’s coherent spaces [GLT89]. (They are also present in some form or the other in most “concrete” semantic models such as concrete sequential algorithms [BC82] and event structures [Win80]). I present a coherent space formulation of the token semantics in Section 4.

To deal with functions, first consider the *linear* functions which use their argument exactly once. (Following Girard [Gir87], We write the linear function space from  $A$  to  $B$  as  $A \multimap B$ .) A linear function has the information  $(a, b)$  if it *uses* the information  $a$  about its input to produce information  $b$  about its output. Thus, tokens for linear functions are pairs  $(a, b)$ . These have the same form as the tokens for  $A \otimes B$ , but we understand that  $a$  is provided *to* the function and  $b$  is produced *by* the function. For mnemonic purpose, I will often write such pairs as  $a \mapsto b$ . As an example, the increment function of type  $int \multimap int$  has information  $i \mapsto i + 1$  for every integer  $i$ .

The general function space  $A \rightarrow B$  is a little more involved. A function might use its argument multiple times and, in each use, it might obtain a different piece of information about its input. Thus, a token for  $A \rightarrow B$  is of the form  $X \mapsto b$  where  $X$  is a finite consistent set of tokens of  $A$ .

Suppose  $F$  is an element of  $Passive[A \rightarrow B]$ , *i.e.*, it is a consistent combination of information tokens about functions from  $A$  to  $B$ .  $F$  uniquely determines a function  $f$  mapping  $Passive[A]$  to  $Passive[B]$  as follows:

$$f(X) = \{ b : \exists (X' \mapsto b) \in F. X' \subseteq X \}$$

Such functions are continuous and stable.  $F$  is often called the *trace* of the function and  $f$  the *graph* of the function.

Here are some examples to make these concepts clear:

program	type	tokens in its meaning
$inc() = \lambda x.x + 1$	$\mathbf{1} \rightarrow (int \multimap int)$	$\{*\} \mapsto (i \mapsto i + 1)$
$twice(f) = \lambda x.f(fx)$	$(A \multimap A) \rightarrow (A \multimap A)$	$\{(i \mapsto i'), (i' \mapsto i'')\} \mapsto (i \mapsto i'')$

The graph of  $inc$  maps  $\emptyset$  to  $\emptyset$  and  $\{*\}$  to the infinite set  $\{(i \mapsto i + 1) : i \text{ an integer}\}$ . The graph of  $twice$  is:

$$f_{twice}(X) = \{(i \mapsto i'') : \exists i'. (i \mapsto i'), (i' \mapsto i'') \in X\}$$

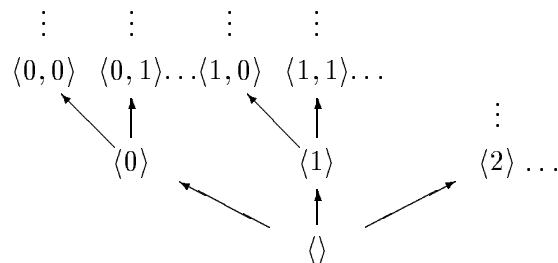
If we have a functional programming language with types  $\tau$ , then each type term determines a semantic type  $A_\tau$  which, in turn, determines a passive domain  $D_\tau$  as follows:

$$\begin{aligned} D_{\mathbf{int}} &= \text{Passive}[int] \\ D_{\tau_1 \times \tau_2} &= \text{Passive}[A_{\tau_1} \times A_{\tau_2}] \cong \text{Passive}[A_{\tau_1}] \times \text{Passive}[A_{\tau_2}] \\ D_{\tau_1 \rightarrow \tau_2} &= \text{Passive}[A_{\tau_1} \rightarrow A_{\tau_2}] \cong \text{Passive}[A_{\tau_1}] \rightarrow_S \text{Passive}[A_{\tau_2}] \end{aligned}$$

where the right hand side constructions are the cpo-product and the stable function space constructions. Since the token space constructions precisely correspond to the cpo constructions, we can usually afford to think solely in terms of cpo's. But, it is not possible to do so when novel type constructions such as linear functions, tensor products (and other type constructions introduced in this paper) need to be modelled.

## 2.1 Active values

Consider a function of type  $A \rightarrow B$  where  $A$  and  $B$  are meant to model *active* values. The function might use its argument multiple times, but it must use it in a *sequential* fashion. The type  $A$  might denote variables or some other higher type values that modify the state. So, each use of the argument might produce *different* behavior. Moreover, the behavior in each use would be dependent on the past behavior. Thus, a token for  $A \rightarrow B$  must be of the form  $(s \mapsto b)$  where  $s$  is a *sequence* of tokens of  $A$ . The sequence denotes the time “history” of the argument as observed by the function. Such histories are naturally ordered by prefix ordering and form a cpo. We call such cpo's *active domains* and write them as  $Active[A]$ . For example, the domain  $Active[int]$  is of the form:



The partial elements of such a domain denote finite approximations of the history of an active object while total elements denote the ideal, (possibly) infinite histories. For instance, a counter “object” that produces successive integers over time has the infinite history  $\langle 0, 1, 2, \dots \rangle$ .

Using histories in the semantics in a naive fashion leads to an *intensional* semantics rather than an extensional one. (All the intermediate states are recorded in the semantics). It is possible to adapt the notion of histories to reflect the input-output relations without mention of intermediate states. This, I relegate to Section 4, and continue to study the intensional version for its simplicity.

A variable of type  $\delta \mathbf{var}$  consists of an  $l$ -value that accepts a  $\delta$ -typed value and an  $r$ -value that produces a  $\delta$ -typed value. So, we define

$$A_{\delta \mathbf{var}} \cong (A_{\delta} \multimap \mathbf{1}) \times A_{\delta}$$

and write the tokens of  $A_{\delta \mathbf{var}}$  as  $i.put$  and  $j.get$  where  $i$  and  $j$  are tokens for  $A_{\delta}$ . (In raw notation, these tokens would be written  $(i \mapsto *) \cdot 1$  and  $j \cdot 2$  respectively). The domain for  $\delta \mathbf{var}$  is  $Active[A_{\delta \mathbf{var}}]$  which has token histories of the form  $\langle i_1.get/put, \dots, i_k.get/put \rangle$ . When a function takes a variable as an input, it may read as well as write the state of that variable and, thereby, develop the history of the variable.

A command gives no interesting output (other than termination). So, we define

$$\begin{aligned} A_{\mathbf{comm}} &= \mathbf{1} \\ D_{\mathbf{comm}} &= Active[\mathbf{1}] \end{aligned}$$

The reader used to thinking of commands as state-to-state functions may find this definition too simplistic. I offer two points by way of explanation. First, conventional models treat states as *static* values whereas we are modelling variables as *active* values. Since active values denote entire histories, both the initial and final states can be represented within a history. Second, notice that  $(A \multimap \mathbf{1}) \multimap \mathbf{1} \cong A$ . So, a command that uses the  $l$ -value of a variable (which is of type  $A_{\delta} \multimap \mathbf{1}$ ) as an *input* is effectively producing a value as an *output*. It is not necessary for such outputs to be reflected in the type of commands themselves.

The following examples illustrate commands at work:

program	tokens in its meaning
$reset : \mathbf{int \ var} \rightarrow \mathbf{comm}$	
$reset(x) = (x := 0)$	$\langle 0.put \rangle \mapsto *$
$reset_1 : \mathbf{int \ var} \otimes \mathbf{int \ var} \rightarrow \mathbf{comm}$	
$reset_1(x, y) = (x := 0)$	$\langle \langle 0.put \rangle, \langle \rangle \rangle \mapsto *$
$inc, tinc : \mathbf{int \ var} \rightarrow \mathbf{comm}$	
$inc(x) = (x := x + 1)$	$\langle i.get, (i + 1).put \rangle \mapsto *$
$tinc(x) = (x := x + 1; x := x + 1)$	$\langle i.get, (i + 1).put, (i + 1).get, (i + 2).put \rangle \mapsto *$

Note that  $reset(x)$  “changes”  $x$  by projecting the *put* component of  $x$  and applying it to integer 0. In a conventional semantics, one would model  $reset(x)$  as a function that takes a “global state” and returns a new state with a modified  $x$  “component”. Our semantics obviates the extraneous notion of global states by directly acting on the history of a variable. The function  $reset_1$  illustrates how a command can ignore a variable (and leave its state unchanged) without using global states.

The meaning of  $tinc$  requires some explanation. Note that the second *get* operation obtains  $i + 1$  (the value used with the previous *put*) and not some arbitrary integer. But, there is nothing in the definition of  $A_{\delta \mathbf{var}}$  that requires it. It is a property of variables allocated using **new** (*regular* variables) that, once we “put” a value in a variable, the next “get” gives back the same value. This motivates the following definition.

**Definition 1** A *regular variable history* (for some type  $\delta$ ) is a sequence  $s$  of tokens of the form  $i.put$  and  $j.get$  (where  $i$  and  $j$  are tokens of  $\delta$ ) such that

1.  $s = \langle \dots, i.put, j.get, \dots \rangle$  implies  $i = j$ .
2.  $s = \langle \dots, i.get, j.get, \dots \rangle$  implies  $i = j$ .

□

In this abstract, we assume that all variables are regular. But, other irregular variables, as in Forsythe [Rey88], are also available in the model.

Next, we must take a closer look at functions. As noted at the outset, a function that operates on active values has tokens of the form  $s \mapsto b$ . These functions are evidently of a different kind from the standard functions of type  $A \rightarrow B$ . To avoid confusion, it is best to use a different notation for the new function space, say, *dynamic function space*,  $A \twoheadrightarrow B$ .<sup>2</sup> Suppose  $F$  is an element of  $Passive[A \twoheadrightarrow B]$ , i.e., it is a consistent combination of information tokens about dynamic functions from  $A$  to  $B$ .  $F$  uniquely determines a function  $f$  mapping  $Active[A]$  to  $Active[B]$  as follows:

$$f(s) = \langle b_1, b_2, \dots \rangle$$

whenever  $s = s_1 \cdot s_2 \cdot \dots$  and  $(s_i \mapsto b_i) \in F$ . Evidently,  $f$  is continuous. It can be verified that it is also stable and that there is cpo-isomorphism:

$$Passive[A \twoheadrightarrow B] \cong Active[A] \rightarrow_S Active[B]$$

As usual, we call  $F$  the *trace* of the dynamic function and  $f$  its *graph*.

The graph of the *reset* program above consists of all input-output pairs of the form

$$\langle 0.put \rangle^k \mapsto \langle * \rangle^k$$

More interesting is the graph of *inc*:

$$\langle i.get, (i+1).put, \dots, (i+k-1).get, (i+k).put \rangle \mapsto \langle * \rangle^k$$

This shows that, if the output of *inc* is used  $k$  times, then a  $k$ -fold repetition of the *get-put* sequence is developed for the input variable's history.

Graphs of functions play an essential role in function composition. (This is the case for ordinary functions as well.) For example, consider the combinator

program	tokens in its meaning
$twice : \mathbf{comm} \rightarrow \mathbf{comm}$	
$twice(c) = (c; c)$	$\langle *, * \rangle \mapsto *$

To find the meaning of the composition  $twice \circ inc$ , we must select from the graph of *inc* all pairs with the output component  $\langle *, * \rangle$ . This gives the trace of the composition to be

$$\langle i.get, (i+1).put, (i+1).get, (i+2).put \rangle \mapsto *$$

Notice how *twice* can affect the “global” state even though its own denotation contains no references to states. For another example, consider a *while* loop combinator defined as follows:

$$\begin{aligned} while & : \mathbf{bool\ exp} \times \mathbf{comm} \rightarrow \mathbf{comm} \\ while & = \mathbf{rec\ } \lambda W. \lambda d. \mathbf{if\ } fst\ d \mathbf{\ then\ } snd\ d; W\ d \mathbf{\ else\ skip} \end{aligned}$$

Its trace, of type  $bool \times \mathbf{1} \twoheadrightarrow \mathbf{1}$ , contains tokens of the form:

$$\langle tt.1, *.2, \dots, tt.1, *.2, ff.1 \rangle \mapsto *$$

---

<sup>2</sup>All the function spaces in this section are dynamic function spaces even though I have not used this notation explicitly.

## 2.2 SCI types

Algol with the SCI type regimen permits three kinds of functions:

$\phi \rightarrow \phi'$  regular functions mapping passive values to passive values, modelled by  $A_\phi \rightarrow A_{\phi'}$ .

$\alpha \rightarrow \alpha'$  dynamic functions mapping active values to active values, modelled by  $A_\alpha \twoheadrightarrow A_{\alpha'}$ .

$\phi \rightarrow \alpha$  regular functions mapping passive values to active values, modelled by  $A_\phi \rightarrow A_\alpha$ .

(The other possible combination  $\alpha \rightarrow \phi$  is prohibited. A little thought reveals that such a type is not very useful.) We have already seen that  $Passive[A_\phi \rightarrow A_{\phi'}] \cong Passive[A_\phi] \rightarrow_S Passive[A_{\phi'}$  and  $Passive[A_\alpha \twoheadrightarrow A_{\alpha'}] \cong Active[A_\alpha] \rightarrow_S Active[A_{\alpha'}]$ . The domain  $Passive[A_\phi \rightarrow A_\alpha]$  is essentially equivalent to  $Passive[A_\phi] \rightarrow_S Passive[A_\alpha]$ , but treated differently. The details of this are not important.

SCI also distinguishes between the so-called passive functions and active functions. Passive functions have been already discussed extensively. Active functions, on the other hand, are not “functions” in the traditional sense. They are history-sensitive objects which exhibit a function-like behavior upon each use, but do not form coherent functions as a whole. For example, a “gensym” (generate symbol) function of type  $string \rightarrow_A string$  **task** may have a history of the kind:

$$\langle \text{“a”} \mapsto \text{task “a0”}, \text{“b”} \mapsto \text{task “b1”}, \text{“a”} \mapsto \text{task “a2”}, \dots \rangle$$

(Think of *task* as some kind of a construction that makes active values out of passive ones.) History-sensitivity is exhibited by giving different outputs for that the same input “a” at different occurrences. The type  $\theta \rightarrow_A \alpha$  represents such history-sensitive functions. The token space for this type is the same as the usual one, but the domain is  $Active[A_{\theta \rightarrow \alpha}]$ .

In general, the domain of a passive type  $\phi$  is  $Passive[A_\phi]$  and the domain of an active type  $\alpha$  is  $Active[A_\alpha]$ . Here is a summary of the semantic types that interpret SCI type terms:

$$\begin{aligned} A_\delta \mathbf{var} &= (A_\delta \multimap \mathbf{1}) \times A_\delta \\ A_\delta \mathbf{exp} &= A_\delta \\ A_{\mathbf{comm}} &= \mathbf{1} \\ A_{\phi \rightarrow \phi'} &= A_\phi \rightarrow A_{\phi'} \\ A_{\alpha \rightarrow \alpha'} &= A_{\alpha \rightarrow_A \alpha'} = A_\alpha \twoheadrightarrow A_{\alpha'} \\ A_{\phi \rightarrow \alpha} &= A_{\phi \rightarrow_A \alpha} = A_\phi \rightarrow A_\alpha \\ A_{\theta \times \theta'} &= A_\theta \times A_{\theta'} \end{aligned}$$

Here,  $\delta$  ranges over primitive types such as **int** and **bool**. My analysis shows that it can also be allowed to range over any discrete (flat) type. Such types include  $\delta \otimes \delta'$  and  $\delta + \delta'$ , but not  $\delta \rightarrow \delta'$  or even  $\delta \multimap \delta'$ .

## 3 Intensional Semantics

I define the semantics of interference-free Algol by induction on type derivations. (The conventional method is to define semantics by induction on syntax. But, using induction on type derivations clarifies type coercions and interference control issues). A phrase  $p$  has a typing of the form

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

where  $x_1, \dots, x_n$  are assumed to be distinct variables and the order of the typing assumptions is immaterial. The meaning of such a phrase is a function of type  $\theta_1^* \otimes \dots \otimes \theta_n^* \multimap \theta$  where  $\theta^*$  is a semantic type defined below. (In this section, we identify term terms with semantic types for simplicity of notation.) As usual, we specify the function in terms of a trace (which is a relation, in general). The traditional *environment* notation is used for the inputs of this function. The empty environment is denoted  $\eta^{(1)}$ ,  $\eta[x \rightarrow a]$  denotes  $\eta$  extended with an  $x$  component having value  $a$ , and  $\eta \cdot \eta'$  denotes the join of two environments  $\eta$  and  $\eta'$  with disjoint domains.

The type  $\theta^*$  is defined as follows:

- $\phi^*$  (for a passive type  $\phi$ ) has finite consistent sets over  $A_\phi$  as its tokens. So,  $\phi^* \multimap \theta$  means the same as  $\phi \rightarrow \theta$ .
- $\alpha^*$  has finite sequences over  $\alpha$  as its tokens. So,  $\alpha^* \multimap \theta$  means the same as  $\alpha \twoheadrightarrow \theta$ . When  $\alpha = \delta \mathbf{var}$ ,  $\alpha^*$  is restricted to contain only regular variable histories.

The insight that  $\phi \rightarrow \theta$  is decomposable into the separate type constructions  $\phi^*$  and  $\multimap$  is central to Girard's linear logic [GLT89]. By using the same idea for the  $\alpha \twoheadrightarrow \theta$  function space as well, we obtain uniformity of notation so that active and passive inputs can be freely intermixed.

We define a generic operation  $seq_\theta$  that maps tokens of  $\theta^* \otimes \theta^*$  to  $\theta^*$  as follows. For passive types,  $seq_\phi(X_1, X_2) = X_1 \cup X_2$  and, for active types,  $seq_\alpha(s_1, s_2) = s_1 \cdot s_2$ . (“ $\cdot$ ” stands for sequence concatenation). We also extend  $seq$  to environments component-wise.

For each type inference rule, we give a semantic derivation rule which states the pairs in the trace of the consequent in terms of those of the antecedents. For the imperative part of the language, the rules are as follows:

$$\begin{array}{c}
\frac{}{\vdash \mathbf{skip} : \mathbf{comm}} \\
\frac{\Gamma \vdash v : \delta \mathbf{var} \quad \Gamma \vdash e : \delta \mathbf{exp}}{\Gamma \vdash v := e : \mathbf{comm}} \\
\frac{\Gamma \vdash c_1 : \mathbf{comm} \quad \Gamma \vdash c_2 : \mathbf{comm}}{\Gamma \vdash (c_1; c_2) : \mathbf{comm}} \\
\frac{\Gamma \vdash c_1 : \mathbf{comm} \quad \Delta \vdash c_2 : \mathbf{comm}}{\Gamma, \Delta \vdash (c_1 \parallel c_2) : \mathbf{comm}} \\
\frac{\Gamma, x : \delta \mathbf{var} \vdash c : \mathbf{comm}}{\Gamma \vdash \mathbf{new } x. c : \mathbf{comm}} \\
\frac{\Gamma, x : \delta \mathbf{exp} \vdash p : \theta}{\Gamma, x : \delta \mathbf{var} \vdash p : \theta} \\
\frac{}{\eta^{(1)} \mapsto *} \\
\frac{\eta \mapsto i.put \quad \eta' \mapsto i}{seq(\eta', \eta) \mapsto *} \\
\frac{\eta \mapsto * \quad \eta' \mapsto *}{seq(\eta, \eta') \mapsto *} \\
\frac{\eta \mapsto * \quad \eta' \mapsto *}{\eta \cup \eta' \mapsto *} \\
\frac{}{\eta[x \rightarrow s] \mapsto *} \\
\eta \mapsto * \\
\frac{\eta[x \rightarrow \emptyset] \mapsto a \quad \eta[x \rightarrow \{i\}] \mapsto a}{\eta[x \rightarrow \langle \rangle] \mapsto a \quad \eta[x \rightarrow \langle i.get \rangle] \mapsto a}
\end{array}$$

**Example 2** We first illustrate the semantic definition by deriving meanings for the simple examples of Section 2. For the *inc* command, we have

$$\frac{x : \mathbf{int } \mathbf{exp} \vdash x + 1 : \mathbf{int } \mathbf{exp} \quad x = \{i\} \mapsto i + 1}{x : \mathbf{int } \mathbf{var} \vdash x : \mathbf{int } \mathbf{var} \quad x : \mathbf{int } \mathbf{var} \vdash x + 1 : \mathbf{int } \mathbf{exp} \quad x = \langle j.put \rangle \mapsto j.put \quad x = \langle i.get \rangle \mapsto i + 1} \\
\frac{}{x : \mathbf{int } \mathbf{var} \vdash x := x + 1 : \mathbf{comm} \quad x = \langle i.get, j.put \rangle \mapsto * \quad \text{where } j = i + 1}$$

For *twice*:

$$\frac{c : \mathbf{comm} \vdash c : \mathbf{comm} \quad c : \mathbf{comm} \vdash c : \mathbf{comm}}{c : \mathbf{comm} \vdash c; c : \mathbf{comm}} \quad \frac{c = \langle * \rangle \mapsto * \quad c = \langle * \rangle \mapsto *}{c = \langle *, * \rangle \mapsto *}$$

**Example 3** For the first equivalence mentioned in introduction, consider the command:

$$p : \mathbf{comm} \rightarrow_A \mathbf{comm} \vdash \mathbf{new } x. x := 0; p(x := x + 1) : \mathbf{comm}$$

The tokens of  $\mathbf{comm} \rightarrow_A \mathbf{comm}$  are of the form  $\langle * \rangle^k \mapsto *$ . Given such a token,  $x := 0; p(x := x + 1)$  executes a sequence of demands on  $x$  of the following form:

$$\langle 0.put, 0.get, 1.put, \dots, (k-1).get, k.put \rangle$$

By the semantics of **new**, the overall phrase gets the semantics:

$$p = \langle \langle * \rangle^k \mapsto * \rangle \mapsto *$$

This is precisely the semantics of  $p(\mathbf{skip})$ . So, the equivalence is valid.

**Example 4** For the second example mentioned in the introduction, consider

$$x : \mathbf{int } \mathbf{var}, f : \mathbf{int } \mathbf{exp} \rightarrow \mathbf{int } \mathbf{exp} \vdash \mathbf{if } x = 0 \mathbf{ then } f(x) \mathbf{ else } 1 : \mathbf{int } \mathbf{exp}$$

The only tokens for  $\mathbf{int } \mathbf{exp} \rightarrow \mathbf{int } \mathbf{exp}$  are  $\emptyset \mapsto j$  and  $\{i\} \mapsto j$ . So, the semantics of the phrase consists of:

$$\begin{aligned} x = \langle 0.get \rangle, f = \{\emptyset \mapsto j\} &\mapsto j \\ x = \langle 0.get, 0.get \rangle, f = \{\{0\} \mapsto j\} &\mapsto j \\ x = \langle i.get \rangle, f = \emptyset &\mapsto 1 \quad \text{for } i \neq 0 \end{aligned}$$

Notice that the semantics of

$$x : \mathbf{int } \mathbf{var}, f : \mathbf{int } \mathbf{exp} \rightarrow \mathbf{int } \mathbf{exp} \vdash \mathbf{if } x = 0 \mathbf{ then } f(0) \mathbf{ else } 1$$

is very similar:

$$\begin{aligned} x = \langle 0.get \rangle, f = \{\emptyset \mapsto j\} &\mapsto j \\ x = \langle 0.get \rangle, f = \{\{0\} \mapsto j\} &\mapsto j \\ x = \langle i.get \rangle, f = \emptyset &\mapsto 1 \quad \text{for } i \neq 0 \end{aligned}$$

The only difference between the two traces is the number of get operations on the variable  $x$  in the second case. Will see that this difference disappears in the extensional version of the semantics (Section 4).

On the other hand, notice that the procedure  $f$  has no capability to change the state of  $x$ . Its parameter is an integer *value*; not an integer variable. In a global store-based semantics, including the most advanced parametricity model [OT93], we are forced to let  $f$  access the  $x$  component of the global store. Thus,  $f$  can temporarily change the state of  $x$  and the equivalence breaks.

**Example 5** Consider the definition of a counter:

$$\begin{aligned} \mathbf{new } x. x := 0; \\ \mathbf{let } \mathbf{counter} = \lambda a. x := x + 1; a \mathbf{ in } \dots \end{aligned}$$

The overall phrase is a command and *counter* is of type  $(\mathbf{int } \mathbf{exp} \rightarrow_A \mathbf{comm}) \rightarrow_A \mathbf{comm}$ . We call the argument  $\mathbf{int } \mathbf{exp} \rightarrow_A \mathbf{comm}$  an “acceptor” and this style of definition “acceptor-passing style”. (Monad-style computations [ORH93, Wad92] give a simpler alternative to acceptor-passing style, but I will not digress into that). We abbreviate a type of the form  $(A \rightarrow_A \mathbf{comm}) \rightarrow_A \mathbf{comm}$  to  $A$  **task**. Its tokens are of the form  $\langle X \mapsto * \rangle \mapsto *$  which we abbreviate to *task X*.

Notice that SCI guarantees that  $x$  is not used in the body of **let**. So, it is completely “captured” inside the counter. We derive the meaning of *counter* as follows. (For simplicity, we ignore the case where  $a$  ignores its argument). The definition of *counter* gets the following trace:

$$x = \langle i.get, (i+1).put, (i+1).get \rangle \mapsto \text{task } \{i+1\}$$

The function graph of this trace is the map:

$$x = \langle i.get, (i+1).put, (i+1).get, \dots, (i+k).get \rangle \mapsto \langle \text{task } \{i+1\}, \dots, \text{task } \{i+k\} \rangle$$

Since the above history of  $x$  is eventually joined with  $\langle 0.put \rangle$ , the useful part of the graph is:

$$x = \langle 0.get, 1.put, 1.get, \dots, k.get \rangle \mapsto \langle \text{task } \{1\}, \dots, \text{task } \{k\} \rangle$$

One can encapsulate the local variable of counter in an acceptor-passing procedure as follows:

```
mkcounter : int exp task task
mkcounter = λa'. new x. x := 0;
             a' (λa. x := x + 1; a x)
```

The meaning of *mkcounter* is simply the trace consisting of

$$\eta^{(1)} \mapsto \text{task } \langle \text{task } \{1\}, \dots, \text{task } \{k\} \rangle$$

for all  $k \geq 0$ . Notice the close similarity with the method of state-encapsulation by streams often used in functional programming.

Consider another version of the counter with a different representation:

```
mkcounter = λa'. new x. x := 0;
             a' (λa. x := x - 1; a (-x))
```

By repeating the above calculation for this program, it may be verified that the meaning of *mkcounter* is still the same. Thus, it may be seen that the history-based semantics is not sensitive to data representations. Modelling such insensitivity is a good part of the theory of [OT93].

The remainder of the semantic definition of Algol/SCI is as follows. I assume that all free variables of passive phrases are passive. (There are indeed various unresolved problems with how best to treat the free variable issues in SCI. See [O’H91, Rey89] for some discussion.) The semantics of function types is simple enough:

$$\frac{\Gamma, x : \theta \vdash p : \theta'}{\Gamma \vdash \lambda x. p : \theta \rightarrow \theta'} \quad (\text{where } \Gamma \text{ is passive}) \quad \frac{\eta[x \rightarrow a] \mapsto b}{\eta \mapsto (a \mapsto b)}$$

$$\frac{\Gamma \vdash p : \theta \rightarrow \theta'}{\Gamma, x : \theta \vdash px : \theta'} \quad \frac{\eta \mapsto (a \mapsto b)}{\eta[x \rightarrow a] \mapsto b}$$

The same semantics applies to all three kinds of function spaces. The semantics of active functions is also the same. The uniformity is achieved by the trick, due to Launchbury [Lau93], of splitting the usual type rule for functional application into two parts: one part as above and a second part called *Compose* below. This trick is reminiscent of sequent calculus as well as categorical logic formulations of type rules.

Most of the distinctions between active and passive types show up in interpreting the so-called “structural” rules. To interpret them, we define a few more generic operations on tokens:

$$\begin{aligned} \text{ignore}_\phi &= \emptyset & \text{ignore}_\alpha &= \langle \rangle \\ \text{read}_\phi(a) &= \{a\} & \text{read}_\alpha &= \langle a \rangle \\ \text{dup}_\phi(X, Y) &= X \cup Y \end{aligned}$$

$\text{ignore}_\theta$  gives the trivial token of  $\theta^*$ ,  $\text{read}_\theta$  maps a token of  $\theta$  to a singleton token of  $\theta^*$  and  $\text{dup}_\phi$  simulates duplication of passive values. The structural rules and their interpretations are as follows:

Id	$x : \theta \vdash x : \theta$	$\eta^{(1)}[x \rightarrow \text{read}(a)] \mapsto a$
Project	$\Gamma \vdash p : \theta'$ $\Gamma, x : \theta \vdash p : \theta'$	$\eta \mapsto a$ $\eta[x \rightarrow \text{ignore}] \mapsto a$
Dup	$\Gamma, x : \phi, y : \phi \vdash p : \theta'$ $\Gamma, z : \phi \vdash p[z/x, z/y] : \theta'$	$\eta[x \rightarrow X_1, y \rightarrow X_2] \mapsto a$ $\eta[z \rightarrow X_1 \cup X_2] \mapsto a$
Compose $_\phi$	$\Gamma \vdash p : \phi \quad \Delta, x : \phi \vdash q : \theta'$ $\Gamma, \Delta \vdash q[p/x] : \theta'$	$\eta_1 \mapsto a_1$ $\vdots$ $\eta_k \mapsto a_k \quad \eta'[x \rightarrow \{a_1, \dots, a_k\}] \mapsto b$ $\text{dup}(\eta_1, \dots, \eta_k) \cdot \eta' \mapsto b$
Compose $_\alpha$	$\Gamma \vdash p : \alpha \quad \Delta, x : \alpha \vdash q : \theta'$ $\Gamma, \Delta \vdash q[p/x] : \theta'$	$\eta_1 \mapsto a_1$ $\vdots$ $\eta_k \mapsto a_k \quad \eta'[x \rightarrow \langle a_1, \dots, a_k \rangle] \mapsto b$ $\text{seq}(\eta_1, \dots, \eta_k) \cdot \eta' \mapsto b$

Note that duplication is only available for passive values, not active values. This is the primary mechanism of interference control. In rule  $Compose_\phi$ , the assumption that passive phrases only contain passive free variables ensures that  $\eta_1, \dots, \eta_k$  have only passive components. Therefore,  $\text{dup}(\eta_1, \dots, \eta_k)$  is meaningful.

## 4 Coherent Semantics

In this section, I give a somewhat terse, technical discussion of the token-based semantics. The issue of “consistency” of tokens was so far left untouched. One way of formalizing this is by Girard’s coherent spaces, which (do below). The reader is referred to [GLT89, Gir87] for a fuller discussion.

A coherent space is a set of tokens together with a binary coherence (consistency) relation. The set of tokens of type  $A$  is denoted  $|A|$ . The fact that two tokens  $a_1$  and  $a_2$  are coherent with each other is denoted by  $a_1 \circ a_2 \text{ [mod } A]$ . (If they are also distinct, we write  $a_1 \frown a_2 \text{ [mod } A]$ ). The coherent spaces for the various types are defined as follows.

$$\begin{aligned} |1| &= \{*\} \\ |A \otimes B| &= |A| \times |B| & (a_1, b_1) \circ (a_2, b_2) &\Leftrightarrow a_1 \circ a_2 \wedge b_1 \circ b_2 \\ |A \times B| &= |A| + |B| & a.i \circ a'.j &\Leftrightarrow i \neq j \vee a \circ a' \\ |A \multimap B| &= |A| \times |B| & (a_1, b_1) \frown (a_2, b_2) &\Leftrightarrow a_1 \circ a_2 \Rightarrow b_1 \frown b_2 \\ |!A| &= A_{\text{fin}} & X \circ Y \text{ [mod } !A] &\Leftrightarrow \forall a \in X. \forall b \in Y. a \circ b \\ |A \rightarrow B| &= |!A| \times |B| & (X_1, b_1) \frown (X_2, b_2) &\Leftrightarrow X_1 \circ X_2 \Rightarrow b_1 \frown b_2 \end{aligned}$$

In the above,  $A_{\text{fin}}$  is the set of finite coherent sets over  $|A|$ , *i.e.*, finite elements of  $Passive[A]$ . Note that there are linear functions

$$\begin{aligned} ignore_A : !A \multimap \mathbf{1} & & \emptyset & \mapsto * \\ read_A : !A \multimap A & & \{a\} & \mapsto a \\ dup_A : !A \multimap !A \otimes !A & & X \cup Y & \mapsto (X, Y) \end{aligned}$$

which allow a passive input to be discarded, read or duplicated respectively. These operations have the mathematical structure of a commutative comonoid. The semantic type  $\phi^*$  used in Section 3 is formalized as  $!\phi$ .

For modelling active types, we need two further constructions on coherent spaces

$$\begin{aligned} |A \triangleright B| &= |A| \times |B| & (a_1, b_1) \frown (a_2, b_2) &\Leftrightarrow a_1 \frown a_2 \vee a_1 = a_2 \wedge b_1 \frown b_2 \\ |\dagger A| &= |A|^* & s \frown r [\text{mod } \dagger A] &\Leftrightarrow (s \text{ prefix } r \vee r \text{ prefix } s \vee \\ & & & s = tas' \wedge r = tbr' \wedge a \frown b [\text{mod } A] \\ & & & \text{for some } t, s', r' \in |A|^*, \text{ and } a, b \in |A| \end{aligned}$$

$A \triangleright B$  is a “sequential” version of  $\otimes$ , *i.e.*, the two components of the pair must be used sequentially. If such a sequential pair has information  $(a, b)$ , it means that it can produce the information  $a$  now, but it can produce the information  $b$  only after  $a$  is completely consumed. Note that there are monomorphisms  $A \otimes B \multimap A \triangleright B$  and  $(A \multimap \mathbf{1}) \triangleright B \multimap (A \multimap B)$ . So, this mysterious form of product is able to act like a tensor product as well as a function space!<sup>3</sup>

The coherent space  $\dagger A$  is the repetitive version of  $\triangleright$ . It models the finite elements of  $Active[A]$ . Note that there are linear functions

$$\begin{aligned} ignore_A : \dagger A \multimap \mathbf{1} & & \langle \rangle & \mapsto * \\ read_A : \dagger A \multimap A & & \langle a \rangle & \mapsto a \\ seq_A : \dagger A \multimap \dagger A \triangleright \dagger A & & s_1 \cdot s_2 & \mapsto (s_1, s_2) \end{aligned}$$

These operations have the mathematical structure of a comonoid. The type  $\alpha^*$  used in Section 3 is formalized as  $\dagger\alpha$ . The restriction of  $\delta \mathbf{var}^*$  to regular variable histories is a feature of the semantics of **new**.

Our result is the following: The semantics of the interference-free Algol can be formalized in terms of coherent spaces and linear functions. This semantics is *adequate*, *i.e.*, whenever a closed program of type **comm** terminates, its meaning is “ $(\eta^{(1)}, *)$ ” (in type  $\mathbf{1} \multimap \mathbf{1}$ ).

## 4.1 Extensional semantics

While the above semantics is simple and intuitive, it is intensional. Note this in Example 4. To model extensional histories, we need to use a more sophisticated linear function  $\dagger A \multimap \dagger A \triangleright \dagger A$  for sequencing. This is given by a function  $\hat{\cdot}$  on sequence tokens of  $\dagger A$  which forms a monoid with the empty sequence as the unit. It is explicitly defined for  $\dagger\delta \mathbf{var}$  as follows:

$$\begin{aligned} \langle i.get \rangle \hat{\cdot} \langle i.get \rangle &= \langle i.get \rangle \\ \langle i.get \rangle \hat{\cdot} \langle j.put \rangle &= \langle i.get, j.put \rangle \\ \langle i.put \rangle \hat{\cdot} \langle i.get \rangle &= \langle i.put \rangle \\ \langle i.put \rangle \hat{\cdot} \langle j.put \rangle &= \langle j.put \rangle \end{aligned}$$

<sup>3</sup>The sequential product construction as well as its curious properties were discovered by C. Retoré of Université Paris VII.

Extensional histories include the empty sequence, singleton sequences and sequences obtained by  $\hat{\cdot}$ . Note such sequences are of length at most 2. In particular sequences of the form  $\langle i.get, j.put \rangle$  are similar to state-to-state functions. This definition of sequencing for  $\dagger\delta \mathbf{var}$  induces a corresponding sequencing operation for every program of type  $\dagger A$ . I will present the details in a future paper.

## 5 Conclusion

I have shown that it is possible to give semantics to higher-order imperative languages without involving a notion of a global state. Such semantics seems to have an intuitive “feel” of being abstract, and this is corroborated by the relative ease with which we are able to validate equivalences which have traditionally failed in conventional approaches.

On the other hand, much needs to be done in and fully understanding all the consequences of this approach. The relationship between the history-based model and the traditional state-to-state function model needs to be better understood. We also need to know in what contexts this form of semantics is fully abstract. A promising result in this direction is the recent full abstraction result due to Cartwright and Felleisen [CF92] which shows a variant concrete sequential algorithms to be fully abstract for an observably sequential version of PCF. It is possible to restate the history-based semantics in a sequential algorithms framework, but the issue of its full abstraction is still open.

Another important question has to do with handling interference. It might seem that the results here have nothing to say about programs with interference. On the contrary, I believe that the interference-controlled subset studied here is a *foundation* for the full language (in much the same way as typed lambda calculus is often viewed as the foundation for the untyped lambda calculus). Firstly, note that SCI *does* allow interference; it merely concentrates it in the type of active pairs ( $\alpha \times \theta$  or  $\theta \times \alpha$ ). Notice this in the while loop combinator (Section 2) which takes an active pair with interfering components. Secondly, this device can be used to model uncontrolled interference. The free variables of a term are treated as the components of a large active tuple (an abstract version of a global store). The result is very much a “possible world” semantics, but a simpler one. The details will have to wait for another paper.

An exciting prospect is to design applicative languages for state manipulation based on the approach of histories. I made some efforts in that direction [Red93b], but the result seems somewhat complex because the  $\triangleright$  type constructor introduces some very unconventional features. A firm understanding of sequencing seems to be a prerequisite to any such ventures.

## References

- [Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, 21(8):613–641, August 1978.
- [BC82] G. Berry and P. L. Curien. Sequential algorithms on concrete data structures. *Theoretical Comp. Science*, 20:265–321, 1982.
- [CF92] R. S. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *Nineteenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 328–342, 1992.
- [FW79] D. P. Freidman and D. S. Wise. An approach to fair applicative multiprogramming. In G. Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lect. Notes in Comp. Science*, pages 203–226. Springer-Verlag, 1979.
- [Gir87] J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.

- [GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Univ. Press, Cambridge, 1989.
- [HMT83] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free list free? In *Tenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 245–257. ACM, 1983.
- [KL85] R. M. Keller and G. Lindstrom. Approaching distributed database implementations through functional programming concepts. In *Intl. Conf. on Distributed Computing Systems*. IEEE, May 1985.
- [KM77] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998, 1977.
- [Laf87] Y. Lafont. Linear logic programming. In P. Dybjer, editor, *Proc. Workshop on Programming Logic*, pages 209–220. Univ. of Goteborg and Chalmers Univ. Technology, Goteborg, Sweden, Oct 1987.
- [Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 144–154, 1993.
- [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.
- [O’H91] P. W. O’Hearn. Linear logic and interference control. In D. H. Pitt et. al., 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.
- [OT92] P. W. O’Hearn and R. D. Tennent. Semantics of local variables. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, pages 217–238. Cambridge University Press (London Math. Soc. Lecture Notes Series), Cambridge, England, 1992.
- [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.
- [PW93] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 1993.
- [Red93a] U. S. Reddy. Higher-order functions and state-manipulation in logic programming. In R. Dyckhoff, editor, *Fourth Workshop on Extensions of Logic Programming*, pages 115–126, St. Andrews, Scotland, Mar 1993. St. Andrews University.
- [Red93b] 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).
- [Red93c] U. S. Reddy. A typed foundation for directional logic programming. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming*, volume 660 of *Lect. Notes in Artificial Intelligence*, pages 282–318. Springer-Verlag, 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.
- [Rey88] J. C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie-Mellon University, June 1988.
- [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.
- [Sco82] D. S. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *Intern. Colloq. Automata, Languages. and Programming*, volume 140 of *Lect. Notes in Comp. Science*, pages 577–613. Springer-Verlag, 1982.
- [Sha83] E. Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, ICOT- Institute of New Generation Computer Technology, January 1983. (Reprinted in [Sha87]).
- [Sha87] E. Shapiro. *Concurrent Prolog: Collected Papers*. MIT Press, 1987. (Two volumes).
- [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.
- [ST86] E. Shapiro and A. Takeuchi. Object-oriented programming in Concurrent Prolog. *New Generation Computing*, 4(2):25–49, 1986. (reprinted in [Sha87]).
- [Sto77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [Ten89] R. D. Tennent. Denotational semantics of Algol-like languages. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol II*. Oxford University Press, 1989. (to appear).
- [Ten90] R. D. Tennent. Semantical analysis of specificaiton logic. *Information and Computation*, 85(2):135–162, 1990.
- [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).
- [Wad92] P. Wadler. The essence of functional programming. In *ACM Symp. on Princ. of Program. Lang.*, 1992.
- [Win80] G. Winskel. *Events in Computation*. PhD thesis, Univ. of Edinburgh, 1980.