

Global State Considered Unnecessary: An Introduction to Object-Based Semantics

UDAY S. REDDY

reddy@cs.uiuc.edu

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.

Abstract. Semantics of imperative programming languages is traditionally described in terms of functions on global states. We propose here a novel approach based on a notion of *objects* and characterize them in terms of their observable behavior. States are regarded as part of the internal structure of objects and play no role in the observable behavior. It is shown that this leads to considerable accuracy in the semantic modelling of locality and single-threadedness properties of objects.

Keywords: Imperative programs, Syntactic control of interference, Denotational semantics, Objects.

1. Introduction

Semantics of programs that deal with state (“imperative programs”) is a long-standing research problem in the theory of programming languages. Traditionally, such programs are given semantics in terms of global states. (See, for example, Stoy [74].) A global state is an element of a suitably large set that gives the value of every storage variable at a given instant of execution, e.g., the set of all sequences of data values ($State = Val^*$). The allocation of a new local variable is interpreted as enlarging the sequence by adding a new component, and deallocation is interpreted as shrinking the sequence by deleting the new component.

Commands are interpreted as transformers of global states, i.e., partial functions of type $State \rightarrow State$. Expressions (as in Algol 60) are interpreted as partial functions of type $State \rightarrow Val$. This much seems fairly natural. But, problems begin to surface as soon as we consider functions. A function phrase (“procedure”) of type $s \rightarrow t$ acts on an argument of type s to produce a result of type t . Now, it is perfectly reasonable for the argument to act on a portion of the state to which the procedure has no access (since local variables can be allocated outside the procedure definition). Similarly, it is also possible for the procedure to act on local variables that cannot be accessed by the argument. Yet, the result of the procedure (of type t) acts on the entire global state. It is not possible to disentangle the actions of the procedure and the argument on the global state.

The following program equivalence, adapted from [37], illustrates this issue:

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

Here, the free identifier¹ p is a function from commands to commands, i.e., it is of type $\mathbf{comm} \rightarrow \mathbf{comm}$. (Equivalently, it is a procedure that takes a “parameterless

procedure” as its argument.) The command on the left allocates a local variable x and calls a procedure p with an increment procedure on x as the argument. Note that the procedure p has no direct access to the variable x . After the completion of the procedure call $p(x := x + 1)$, the variable x is deallocated. Since x is never read before it is discarded, the effects of $p(x := x + 1)$ on the variable x are unobservable. Hence, calling the procedure p with a trivial “do nothing” procedure **skip** as the argument should have an equivalent effect.

Now, consider the global state interpretation. The meaning of p must be a function that maps global state transformers to global state transformers, i.e., a function of type $[State \rightarrow State] \rightarrow [State \rightarrow State]$. There is no *a priori* reason for the result of p to respect the local variable abstractions. For instance, p might denote the function

$$p(f) = \lambda s. \begin{cases} s, & \text{if } |s| = n \geq 1 \text{ and } s(n) = 0 \\ \text{undefined,} & \text{otherwise} \end{cases}$$

which ignores its argument and tests if the most recent component of the state is 0. For this p , the left hand side of equivalence (1) evaluates to the do-nothing state transformer but the right hand side gives a state transformer that is typically undefined, in particular, when applied to the empty state. (In place of “undefined,” one can substitute any state transformer with an observable effect such as printing a “.” on the screen.) The problem is that $p(f)$ is a transformer of *global states*. There are infinitely many such functions p which cannot be expressed in programming languages with local variable abstractions.

This failure of the global state semantics has been recognized for some time, and various solutions have been devised for modelling local variable abstractions [23], [37], [48], [49], [51], [66], [78]. (The survey articles [48], [77] and the text [79] give good overviews of the extant techniques.) Essentially, all these solutions view commands not as state transformers, but as families of state transformers indexed by state sets (formalized as *functors* in the categorical sense). Procedures are viewed as families of functions with appropriate uniformity conditions (typically *naturality*). While these solutions lose some of the simplicity of the global state semantics by moving to second-order concepts (like functors and natural transformations), they are still unable to represent local variable abstractions sufficiently to validate equivalence (1).

Another issue that one must address in the semantics of imperative programs is that state changes are actually “changes,” an issue first raised in [50]. Running a command causes the state of the program to be destructively overwritten by a new state, and it is not possible, in general, to revert to the original state. If, for example, the procedure p in (1) runs its argument command, the value of x would get incremented and there is no way for the procedure to revert to the original value of x . The following equivalence, due to P. W. O’Hearn, makes this explicit:

$$\begin{aligned} & (\text{new}[\text{int}] x. x := 0; p(x := x + 1); \text{if } x > 0 \text{ then diverge else skip}) \\ & \equiv p(\text{diverge}) \end{aligned} \quad (2)$$

The equivalence must hold because both sides diverge if p runs its argument. (Again, one can use commands with observable effects in place of **diverge** to create similar examples.) But, state transformers are mathematical functions. They have no notion of “change” built-in. The state transformer representing $p(x := x+1)$ may very well yield the original state even after computing new states. The following function p of type $[State \rightarrow State] \rightarrow [State \rightarrow State]$ represents such a “snap-back” operation:

$$p(f) = \lambda s. \begin{cases} s, & \text{if } f(s) \text{ is defined} \\ \text{undefined,} & \text{otherwise} \end{cases}$$

Thus, the equivalence (2) fails in all known semantic models of imperative programs.²

We believe that most of these difficulties accrue from a premature focus on states. Our intuitions suggest that states have to do with the *internal* structure of implementations, not the observable behavior. For example, in classical automata theory, the internal structure of machines is defined in terms of states, but the observable behavior is defined in terms of languages or sequential functions that the machines recognize or compute. If the meaning of program phrases is similarly specified in terms of observable behavior, we might finesse all the intricate mathematical issues regarding state sets in higher-order programming languages.

The author initiated a fresh approach to the semantics of imperative programs in the summer of 1992 with these considerations. The inspiration came from linear logic [17] and its close connection to Reynolds’s syntactic control of interference [64] as espoused by O’Hearn [44]. Being a radically different approach to the semantics of imperative programs, it could initially be stated only in very theoretical terms [60]. However, it is now possible to explain the approach in the everyday programming terminology of “objects” (with an attendant formalization of this concept).

An *object* is a state machine; it has an internal state and a collection of externally observable operations. Invoking these operations can affect the internal state so that successive observations can produce different results. Thus, objects have a history-sensitive behavior. The key fact is that it is possible to specify the observable behavior of objects without reference to states. Moreover, such object behaviors form domains. Appropriate functions between such domains denote “constructions” for building objects from other objects. Program phrases are interpreted as functions denoting such constructions. This is the essence of what we call the *object-based* approach to the semantics of imperative programs.

Applying these ideas to equivalence (1), we interpret the procedure p as acting on “command objects” (objects with a single operation of type **comm**). The phrase $x := x + 1$ builds a command object from a variable object. The phrase **skip** gives a trivial command object. The key fact is that the two command objects have precisely the same observable behavior: their command operation can be invoked an arbitrary number of times and it terminates each time. (The effect of the invocations on the internal state of the objects differs but this is transparent

to the procedure p .) Therefore, the behavior of the procedure p in the two cases must be equal.

Applying these same ideas to equivalence (2), we note that the procedure p can only use its argument object via its command operation. It has no direct access to the internal state of this object. Since there is no operation provided in the object for reverting its state to an earlier one, state changes caused by invoking its command operation are irreversible.

Our experience shows that the accuracy of the object-based model is comparable to the best state-based models. In fact, the object-based model goes further in modelling the “irreversible change” (or “single-threadedness”) aspect of objects which, so far, has not been possible in the state-based models. The surprising fact is that all this can be done at the “ground level” in terms of domains and functions without moving to second-order concepts such as functors and natural transformations. However, the “ground level” semantics exists only for certain well-behaved programs that we call “interference-controlled” programs.

Interference is the phenomenon of two program phrases acting on a common piece of store. For example the command $x := a$ interferes with the expression $x + 1$ because both act on the common variable x . *Interference-control* is the programming discipline that maintains that interfering phrases should not be used in contexts where they might appear to be independent. For example, in a procedure call $P(Q_1, Q_2)$, written in the traditional Algol notation, one would normally expect the phrases P , Q_1 and Q_2 to be independent. If Q_1 were the command $x := a$ and Q_2 the expression $x + 1$, then the standard reasoning about the procedure P would become invalid.

Conventional formulations of Hoare logic [25], [26] assume freedom from aliasing (which is an instance of interference control). They would become unsound if aliasing were permitted. One way to reason about aliasing and interference is to use a sound generalization of Hoare logic such as Reynolds’s specification logic [49], [67], [65], [78], but most programmers find the overhead of reasoning about interference in this logic to be excessive.

We believe that a similar situation arises with denotational semantics. The global state concepts that occur in the conventional approaches are there in order to handle interference. A better focus on the key concepts is obtained by considering interference-controlled programs. In a sense, the role of interference control is parallel to that of types. Just as typed lambda calculus is often taken to be the foundation underlying the untyped lambda calculus [70], we believe that interference-controlled languages should be treated as the foundation for languages with free interference.

What we seek is a semantics of interference-controlled Algol that has a structure similar to that of functional programming languages. The standard semantics of PCF [56], for example, interprets types as domains and terms as (continuous) functions. We would similarly like a semantics that interprets interference-controlled Algol types as domains and phrases as appropriate functions between domains. It appears that this goal is unlikely to be achieved for full Algol with uncontrolled

interference. On the other hand, such a semantics is very well possible and quite attractive for interference-controlled Algol.³

In this paper, we explain the basic ideas of the object-based model using the semantic framework of *coherent spaces*. Coherent spaces are a particularly simple form of dI-domains [10] formulated by Girard [17], [19]. Among their notable accomplishments is the inspiration for linear logic. In fact, the earliest (denotational) model for linear logic was in terms of coherent spaces. We inherit them by way of linear logic. We would like to stress, however, that coherent spaces form just one framework where the object-based concepts can be formalized. There can be many other frameworks where objects can live. Some recent frameworks that seem to have this capability include [4], [5], [14], [84]. We expect that many of these new frameworks will be used for modelling objects in the future, and some of them, in particular games models, will improve on coherent spaces in accuracy.

Related work

The previous work on semantics of local variables was already mentioned. In recent work, carried out almost in parallel with the present work, O’Hearn and Tennent [50] have used the ideas of relational parametricity to characterize the necessary uniformity conditions for procedures. Sieber [72], [73] presents a similar model in the context of locally complete partial orders. These recent models do validate equivalence (1) and other equivalences having to do with local variables. However, a treatment of irreversible state changes in this framework is still awaited.

The reader would notice that the present semantics runs very much counter to these approaches. While they attempt to restrict a global state-based semantics to meaningful denotations, we start by identifying the observable behaviors of individual state variables and build up to larger values.

The earlier efforts in studying the semantics of interference control include those of Tennent [76] in the traditional global state approach and O’Hearn [43], [45] in the functor category approach. More recently, O’Hearn and Tennent [46] formulated a model which, though in an explicit-state functor category framework, attempts to split up the global state into smaller pieces.

Girard’s linear logic [17] was an important source of new ideas. For pedagogical reasons, we mention little of linear logic in this paper, but the informed reader can see linear logic ideas everywhere. Considerable debt is owed to O’Hearn’s work [44] on relating linear values and active values (as in linear logic and SCI respectively). This work, together with Wadler’s related insights [80] formed the original inspiration for this work. Another important contribution from the linear logic side is the “before” connective, due to Girard and Retoré [63], which played a prominent role in the early development of this model [60], [59].

There is a large body of work on concurrency which subscribes to essentially the same philosophy as that advocated here: state is localized in processes and any notion of global state is shunned. Some of the major pieces of work in this area include CSP [28], CCS [39] and actors [6]. In relation to concurrency, it should be

noted that, in studying imperative programming, we deal with *objects* instead of processes. The difference is that objects lack any form of internal “control”. This is a simplification but, at the same time, results in much structure that is hard to find in process calculi, such as functions, types and higher-order values. Our focus will be on modelling this structure. However, the author has, after the fact, become aware of many commonalities between the present semantic model and models of concurrency, and has adopted the terminology of concurrency for related notions, such as *traces* [28], [35] and *pomsets* [57]. Some of the recent work in concurrency is moving towards semantic concepts like functions and types [22], [24], where again some similarities with the current work become apparent. Abramsky has been studying concurrency using linear logic foundations (see, e.g., [2]), leading to similar notions as ours. However, unlike him, we choose to adapt the conventional denotational paradigm to the issues of state instead of replacing it with a new one.

Another important piece of related work is Shapiro’s modelling of state in concurrent logic programming languages [71]. This is closely related to the classical stream-based approach to modelling state in functional programming [16], [31], [32]. However, concurrent logic programming languages fill in an important missing piece in this set-up (often called “logic variables”). Our model may be seen as a formalization of the dependencies implicit in the use of logic variables.

Since we use concepts of “objects” pervasively in this work, one might wonder if there is some link with the semantic models of object-oriented programs. It is not yet clear if such is the case. The majority of the semantics work in object-oriented programming seems focused on unraveling inheritance. The issues of state are rarely addressed. The work that does address state, such as [62], uses the traditional state-based approach. Some of the work on inheritance [54], however, suggests that the issues of state and inheritance might indeed be related.

Overview

We first review the language of interference-controlled Algol, which forms the subject of this study (Section 2). In Section 3, we introduce the basic concepts of object-based semantics informally and motivate the issues. Sections 4 and 5 define the formal aspects of the model. Section 6 is devoted to a brief study of the semantic domains obtained. In Section 7, we illustrate the semantics by reasoning about several example equivalences. Section 8 briefly reviews some possible extensions to the language and semantic framework.

In Appendix A, we gather together the salient mathematical structure of the model in a categorical framework. Appendix B contains a proof of computational adequacy of the semantics.

2. Interference-controlled Algol

Idealized Algol, due to Reynolds [66], 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:

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

Thus, the type system of Algol is described by the following syntax of types:

$$\begin{aligned} \text{data types } \delta &::= \mathbf{int} \mid \mathbf{bool} \mid \dots \\ \text{(phrase) types } \theta &::= \mathbf{var}[\delta] \mid \mathbf{exp}[\delta] \mid \mathbf{comm} \mid \theta_1 \times \theta_2 \mid \theta_1 \rightarrow \theta_2 \end{aligned}$$

Though Algol 60 [41] is one of the oldest programming languages, it has concepts that are remarkably modern. In particular, it is the only widely known imperative programming language that satisfies unrestricted β equivalence (often identified with “referential transparency”). Several recent proposals for integrating functional and imperative programming features [42], [53], [75] involve ideas resembling those of Algol, and Reynolds himself has been designing a successor language called Forsythe [68]. Therefore, we take Algol to be the focus of our study. The reader should be able to relate these concepts to other programming languages. (In particular, see the discussion in Sec. 8.)

Syntactic control of interference

The central idea of syntactic control of interference (SCI), also due to Reynolds [64], [69], is that, in a function application $(P Q)$, the phrases P and Q should be “independent”, i.e., P does not change something that Q reads or writes and *vice versa*. Ensuring that applications always satisfy independence has the benefit that all free identifiers are always independent. Thus, to check for the independence of P and Q , one only needs to check that they have no common free identifiers.

However, imposing that P and Q never have common free identifiers is too stringent. For example, we could not write $plus\ x\ x$ where $plus$ is of type $\mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{exp}[\mathbf{int}]$. To relax the restriction, Reynolds identifies a special class of values called *passive* values which never change the state. Passive values are independent of each other, including themselves. Free identifiers denoting passive values can then be shared by P and Q . Since x in $plus\ x\ x$ is passive, such an application is permitted.

To accord special treatment to passive values, we identify a subclass of types called *passive types*:

$$\text{passive types } \phi ::= \mathbf{exp}[\delta] \mid \phi_1 \times \phi_2 \mid \theta \rightarrow \phi \mid \theta_1 \rightarrow_p \theta_2$$

Table 1. Type syntax of interference-controlled Algol

$$\begin{array}{c}
\frac{}{\Pi \mid \Gamma, x : \theta \vdash x : \theta} \text{Id} \\
\frac{\Pi \mid \Gamma, x : \theta \vdash P : \theta'}{\Pi \mid \Gamma \vdash \lambda x. P : \theta \rightarrow \theta'} \rightarrow \mathcal{I} \quad \frac{\Pi_1 \mid \Gamma_1 \vdash P : \theta \rightarrow \theta' \quad \Pi_2 \mid \Gamma_2 \vdash Q : \theta}{\Pi_1, \Pi_2 \mid \Gamma_1, \Gamma_2 \vdash P Q : \theta'} \rightarrow \mathcal{E} \\
\frac{\Pi \mid \Gamma \vdash P : \theta_1 \quad \Pi \mid \Gamma \vdash Q : \theta_2}{\Pi \mid \Gamma \vdash (P, Q) : \theta_1 \times \theta_2} \times \mathcal{I} \quad \frac{\Pi \mid \Gamma \vdash P : \theta_1 \times \theta_2}{\Pi \mid \Gamma \vdash \text{fst}(P) : \theta_1} \times 1 \mathcal{E} \quad \frac{\Pi \mid \Gamma \vdash P : \theta_1 \times \theta_2}{\Pi \mid \Gamma \vdash \text{snd}(P) : \theta_2} \times 2 \mathcal{E} \\
\frac{\Pi, x_1 : \theta, x_2 : \theta \mid \Gamma \vdash P : \theta'}{\Pi, x : \theta \mid \Gamma \vdash [x/x_1, x/x_2]P : \theta'} \text{Contr} \\
\frac{\Pi, x : \theta \mid \Gamma \vdash P : \theta'}{\Pi \mid \Gamma, x : \theta \vdash P : \theta'} \text{Activate} \quad \frac{\Pi \mid \Gamma, x : \theta \vdash P : \phi}{\Pi, x : \theta \mid \Gamma \vdash P : \phi} \text{Passify} \\
\frac{\Pi \mid \vdash P : \theta_1 \rightarrow \theta_2}{\Pi \mid \vdash P : \theta_1 \rightarrow_p \theta_2} \text{Promote} \quad \frac{\Pi \mid \Gamma \vdash P : \theta_1 \rightarrow_p \theta_2}{\Pi \mid \Gamma \vdash P : \theta_1 \rightarrow \theta_2} \text{Derelict}
\end{array}$$

According to this syntax, expressions are passive, pairs of passive values are passive and all functions returning passive values are passive. In addition, a special class of functions called *passive functions* (whose type is written with the \rightarrow_p constructor) are passive. Passive functions are those that do not modify global variables.

Adding passivity to the SCI type regimen leads to certain technical problems with regard to subject reduction, as noted in [64]. An elegant solution to these problems was recently formulated by O’Hearn *et al.* [46]. We adopt their solution below.

Syntax

The terms of an imperative programming language are often called *phrases* in order to avoid confusion with *expressions* (which are phrases of the specific type $\mathbf{exp}[\delta]$). The (unchecked) phrases of interference-controlled Algol have the following context-free syntax:

$$P ::= x \mid k \mid \lambda x. P \mid P_1 P_2 \mid (P_1, P_2) \mid \text{fst}(P) \mid \text{snd}(P)$$

where k ranges over constants. This syntax is the same as that of any (applied) lambda calculus, but the novelty lies in the type syntax, shown in Table 1.

A typing judgement is of the form

$$\Pi \mid \Gamma \vdash P : \theta$$

where

- P is a phrase,
- θ is a type,
- $\Pi = \{x_1 : \theta_1, \dots, x_n : \theta_n\}$ and $\Gamma = \{y_1 : \theta'_1, \dots, y_m : \theta'_m\}$ are *sets* of typing assumptions (normally written without braces) for the free identifiers of P , such that
- $x_1, \dots, x_n, y_1, \dots, y_m$ are *distinct* identifiers.

A judgement of this form may be read as “ P is a well-typed phrase of type θ using the identifiers $x_1 : \theta_1, \dots, x_n : \theta_n$ passively and the identifiers $y_1 : \theta'_1, \dots, y_m : \theta'_m$ possibly actively.” The type contexts Π and Γ are respectively called the “passive zone” and the “active zone” of the typing judgement. The identifiers occurring in them are called the “passive free identifiers” and “active free identifiers” of P .

The first three lines of Table 1 form what may be called an “affine” lambda calculus (with product types). Its distinguishing feature is that, in forming a function application phrase (PQ), we have to split the available free identifiers into two disjoint sets $\Pi_1 \mid \Gamma_1$ and $\Pi_2 \mid \Gamma_2$ and use them in P and Q respectively. This means that P and Q cannot share free identifiers. This implements Reynolds’s principle of interference control.

Note that no such identifier restrictions are applicable to pair formation (P, Q). This means that the two components of a pair could be *interfering* in general. It also means that the currying transformation is not applicable: the types $\theta_1 \times \theta_2 \rightarrow \theta'$ and $\theta_1 \rightarrow \theta_2 \rightarrow \theta'$ have different meanings. It is possible to add a separate type constructor for “independent products” which would put together non-interfering components. We consider this briefly in Sec. 8.

The remaining rules are referred to as *structural rules*. They are concerned with maintenance of identifiers and types. The rule *Contr* (for “contraction”) on the fourth line brings back limited amount of sharing. (We use the notation $[Q/x]P$ for the substitution of Q for all occurrences of x in P .) Any identifier x occurring in the passive zone can be made into two copies x_1 and x_2 , which can then be used in the two branches of a function application phrase. This allows us to write expression phrases like *plus x x*, for example.

The two rules *Activate* and *Passify* let identifiers move between zones. The rule *Activate* says that any passive free identifier can be regarded as an active free identifier. (This is necessary for binding it by the rule $\rightarrow \mathcal{I}$.) The rule *Passify* says that any active free identifier can be regarded as a passive free identifier as long as it is used in a *passive* phrase. (Note that this is the only place in the type system where passive types are distinguished from other types.) This allows us to derive, for example:

$$v : \mathbf{var}[\mathbf{int}] \mid \vdash \mathbf{deref } v : \mathbf{exp}[\mathbf{int}]$$

Even though the identifier v is of an active type $\mathbf{var}[\mathbf{int}]$, it is used passively in $\mathbf{deref } v$ because the latter is of a passive type $\mathbf{exp}[\mathbf{int}]$. This curious form of the

Table 2. Primitive phrases of interference-controlled Algol

$\Pi \mid \Gamma \vdash 0 : \mathbf{exp[int]}$	$\frac{\Pi \mid \Gamma \vdash E_1 : \mathbf{exp[int]} \quad \Pi \mid \Gamma \vdash E_2 : \mathbf{exp[int]}}{\Pi \mid \Gamma \vdash E_1 + E_2 : \mathbf{exp[int]}}$	$\Pi \mid \Gamma \vdash \mathbf{skip} : \mathbf{comm}$
$\Pi \mid \Gamma \vdash C_1 : \mathbf{comm} \quad \Pi \mid \Gamma \vdash C_2 : \mathbf{comm}$	$\Pi_1 \mid \Gamma_1 \vdash C_1 : \mathbf{comm} \quad \Pi_2 \mid \Gamma_2 \vdash C_2 : \mathbf{comm}$	
$\Pi \mid \Gamma \vdash C_1; C_2 : \mathbf{comm}$		$\Pi_1, \Pi_2 \mid \Gamma_1, \Gamma_2 \vdash C_1 \parallel C_2 : \mathbf{comm}$
$\Pi \mid \Gamma, x : \mathbf{var}[\delta] \vdash C : \mathbf{comm}$	$\Pi \mid \Gamma \vdash V : \mathbf{var}[\delta] \quad \Pi \mid \Gamma \vdash E : \mathbf{exp}[\delta]$	$\Pi \mid \Gamma \vdash V : \mathbf{var}[\delta]$
$\Pi \mid \Gamma \vdash \mathbf{new}[\delta] x. C : \mathbf{comm}$	$\Pi \mid \Gamma \vdash V := E : \mathbf{comm}$	$\Pi \mid \Gamma \vdash \mathbf{deref} V : \mathbf{exp}[\delta]$
$\Pi \mid \Gamma \vdash E : \mathbf{exp[bool]} \quad \Pi \mid \Gamma \vdash P_1 : \theta \quad \Pi \mid \Gamma \vdash P_2 : \theta$		$\Pi \mid x : \theta \vdash P : \theta$
$\Pi \mid \Gamma \vdash \mathbf{if} E \mathbf{then} P_1 \mathbf{else} P_2 : \theta$		$\Pi \mid \vdash \mathbf{rec} x. P : \theta$

Passify rule has a direct correspondence with the semantics described in Sec. 5.⁴ Since v is regarded as a passive free identifier, one can use *Contr* to further derive

$$v : \mathbf{var[int]} \mid \vdash \mathbf{plus} (\mathbf{deref} v) (\mathbf{deref} v) : \mathbf{exp[int]}$$

Finally, the last two rules allow a function phrase to be given a passive function type provided it has *no active free identifiers* and to forget the fact that a function phrase is passive.

The following is a sample of constants one finds in an Algol-like language.

$$\begin{aligned}
0 & : \mathbf{exp[int]} \\
+ & : \mathbf{exp[int]} \times \mathbf{exp[int]} \rightarrow_p \mathbf{exp[int]} \\
\mathbf{skip} & : \mathbf{comm} \\
; & : \mathbf{comm} \times \mathbf{comm} \rightarrow_p \mathbf{comm} \\
\parallel & : \mathbf{comm} \rightarrow_p (\mathbf{comm} \rightarrow \mathbf{comm}) \\
\mathbf{new}[\delta] & : (\mathbf{var}[\delta] \rightarrow \mathbf{comm}) \rightarrow_p \mathbf{comm} \\
:=_\delta & : \mathbf{var}[\delta] \times \mathbf{exp}[\delta] \rightarrow_p \mathbf{comm} \\
\mathbf{deref}_\delta & : \mathbf{var}[\delta] \rightarrow_p \mathbf{exp}[\delta] \\
\mathbf{if}_\theta & : \mathbf{exp[bool]} \times \theta \times \theta \rightarrow_p \theta \\
\mathbf{rec}_\theta & : (\theta \rightarrow_p \theta) \rightarrow_p \theta
\end{aligned}$$

For convenience and clarity, we associate with them the concrete syntax shown in Table 2. The parallel composition combinator $C_1 \parallel C_2$ is added to show the analogy with function application: the two subcommands C_1 and C_2 cannot share active free identifiers. The construct $\mathbf{new}[\delta] x. C$ creates a local variable and binds it to the identifier x within the scope of C . Since it is a binding construct, it is formally treated as a second-order function. We often omit writing the combinator \mathbf{deref} except for clarity in some examples. We use \mathbf{undef} as a short form for the diverging phrase $\mathbf{rec} x. x$.

Discussion

If we disregard the higher types such as $\theta_1 \times \theta_2$ and $\theta_1 \rightarrow \theta_2$, we obtain the language of basic imperative programs. This language is essentially defined by the combinators of Table 2. Its programs have standard semantics in terms of state transformation (or *execution*). When we form a command using higher-type abstractions, the semantics of lambda calculus becomes applicable. The command phrase is interpreted by first *reducing* it, using the standard reduction rules such as β -reduction, until one obtains a command phrase that can be executed.⁵ Thus, the semantics of Idealized Algol is an orthogonal combination of typed lambda calculus and the language of basic imperative programs [66], [81].

As a simple example, consider the function

$$\begin{aligned} \textit{twice} &: \mathbf{comm} \rightarrow \mathbf{comm} \\ \textit{twice } c &= c; c \end{aligned}$$

When applied to a command c , the function produces a command that has the effect of running c twice. For example, $\textit{twice} (\textit{print } '*')$ produces a command that prints two $*$'s. It is important to note that the function \textit{twice} itself does not “run” any commands. It is an honest-to-goodness function that produces a command as its result. This result must then be used in a context where it will be executed to produce any effect. If it is used in a context where it is ignored, no effects are produced. For example, if \textit{ignore} is the function $\lambda c. \mathbf{skip}$, then $\textit{ignore} (\textit{twice} (\textit{print } '*'))$ produces no effect even when it is run.

As an aside, note that \textit{twice} can be given a passive function type:

$$\textit{twice} : \mathbf{comm} \rightarrow_p \mathbf{comm}$$

Since it has no active free identifiers, it cannot cause any state modifications to nonlocal variables.

As a generalization of \textit{twice} , consider a function that maps a command c to an n -fold composition of c with itself:

$$\begin{aligned} \textit{times} &: \mathbf{exp}[\mathbf{int}] \rightarrow_p \mathbf{comm} \rightarrow_p \mathbf{comm} \\ \textit{times } n \ c &= \mathbf{if } n = 0 \ \mathbf{then } \mathbf{skip} \\ &\quad \mathbf{else } c; \textit{times } (n - 1) \ c \end{aligned}$$

If n is any natural number, $\textit{times } n$ is a function of type $\mathbf{comm} \rightarrow_p \mathbf{comm}$. This shows that every natural number can be encoded as a function of type $\mathbf{comm} \rightarrow_p \mathbf{comm}$. It turns out that the converse is true as well, i.e., every function of type $\mathbf{comm} \rightarrow_p \mathbf{comm}$, other than the undefined function, is equivalent to $\textit{times } n$, for some natural number n . Thus, functions of type $\mathbf{comm} \rightarrow_p \mathbf{comm}$ are similar to Church numerals.

Control structures for commands can be defined as functions on commands. For example, the function:

$$\begin{aligned}
& \mathit{while} : \mathbf{exp}[\mathbf{bool}] \times \mathbf{comm} \rightarrow_p \mathbf{comm} \\
& \mathit{while} (b, c) = \mathbf{if} \ b \ \mathbf{then} \ c; \ \mathit{while} (b, c) \\
& \qquad \qquad \mathbf{else} \ \mathbf{skip}
\end{aligned}$$

defines a while loop control structure. Novel control structures can be defined just as easily. For example, by adding recursive data types like lists, we may define higher-order operations for commands analogous to *map*:

$$\begin{aligned}
& \mathit{mapdo} : (\theta \rightarrow \mathbf{comm}) \rightarrow_p (\mathbf{list} \ \theta \rightarrow \mathbf{comm}) \\
& \mathit{mapdo} \ f \ [] = \mathbf{skip} \\
& \mathit{mapdo} \ f \ (x :: xs) = f \ x; \ \mathit{mapdo} \ f \ xs \\
& \mathit{listdo} : \mathbf{list} \ \mathbf{comm} \rightarrow_p \ \mathbf{comm} \\
& \mathit{listdo} \ [] = \mathbf{skip} \\
& \mathit{listdo} \ (c :: cs) = c; \ \mathit{listdo} \ cs
\end{aligned}$$

Note that we have $\mathit{mapdo} \ f = \mathit{listdo} \circ (\mathit{map} \ f)$ as an equational law of these functions.

More important to our concerns is the ability to build *objects* with internally encapsulated state and exported operations (called *methods*). Such objects are typically built from variables (which are themselves primitive objects built-in by the language). For example, the following function builds counter objects from variables:

$$\begin{aligned}
& \mathbf{counter} = \mathbf{exp}[\mathbf{int}] \times \mathbf{comm} \\
& \mathit{mkcounter} : \mathbf{var}[\mathbf{int}] \rightarrow_p \ \mathbf{counter} \\
& \mathit{mkcounter} \ v = (\mathbf{deref} \ v, \ v := v + 1)
\end{aligned}$$

A counter object has two methods: the first, of type $\mathbf{exp}[\mathbf{int}]$, returns the current value of the counter and the second, of type \mathbf{comm} , increments the value of the counter.

We often use sugared *record notation* for product types so that mnemonic names for the fields are available. In this notation, we write *mkcounter* as:

$$\begin{aligned}
& \mathbf{counter} = [\mathbf{val} : \mathbf{exp}[\mathbf{int}] \times \mathbf{inc} : \mathbf{comm}] \\
& \mathit{mkcounter} : \mathbf{var}[\mathbf{int}] \rightarrow_p \ \mathbf{counter} \\
& \mathit{mkcounter} \ v = [\mathbf{val} = \mathbf{deref} \ v, \ \mathbf{inc} = (v := v + 1)]
\end{aligned}$$

We would then write $c.\mathbf{val}$ and $c.\mathbf{inc}$ for the methods of a counter c .

To build and use a counter object in a larger program, we use one of two techniques. By the first technique, we might use an existing variable v to build a counter object, bind the counter to an identifier c , and then use c in a command. This corresponds to a program structure of the following form:

$$\begin{aligned}
& \mathbf{let} \ c = \mathit{mkcounter} \ v \\
& \mathbf{in} \ P
\end{aligned}$$

where P is a command phrase. The let-block is desugared in the standard fashion: $(\lambda c. P)(\mathit{mkcounter} \ v)$. In this form of a program, we must think of the variable v as

the “raw material” used for building the counter c . While the counter is in use, we cannot directly access this raw material. Doing so would cause interference. Indeed, SCI ensures that v does not occur free in the phrase P . However, the variable v can be used in the commands that precede or follow the let-block. In this sense, *mkcounter* builds a temporary “scaffolding” around the variable v which is in effect during the execution of P .

The second technique is to create a *new variable* and erect a permanent scaffolding around it in the form of a counter. This is expressed by the following function:

$$\begin{aligned} \mathit{newcounter} &: (\mathbf{counter} \rightarrow \mathbf{comm}) \rightarrow_p \mathbf{comm} \\ \mathit{newcounter} \ k &= \mathbf{new}[\mathbf{int}] \ v. \\ &\quad v := 0; \\ &\quad k \ (\mathit{mkcounter} \ v) \end{aligned}$$

This function takes as its parameter a counter “consumer” k . It creates and initializes a new variable v and supplies the consumer with a counter built from v . To create and use a counter object, we use a program structure of the form:

$$\mathit{newcounter} \ \lambda c. \\ P$$

where P is a command phrase. This corresponds to what would be written as

$$\{\mathbf{counter} \ c; P\}$$

in a more traditional object-oriented programming language. The function *newcounter* corresponds to a “class”. Its task is to create new instance objects of its class.

We close this discussion with a longer example that puts many of these ideas together. A “histogram object” is an array of counters, i.e., a counter-valued function on a finite range of integers, while a histogram is an array of integers:

$$\begin{aligned} \mathbf{histobj} &= \mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{counter} \\ \mathbf{histogram} &= \mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{exp}[\mathbf{int}] \end{aligned}$$

We can create new histogram objects using:

$$\begin{aligned} \mathit{new_hist} &: \mathbf{exp}[\mathbf{int}] \times \mathbf{exp}[\mathbf{int}] \rightarrow_p (\mathbf{histobj} \rightarrow \mathbf{comm}) \rightarrow_p \mathbf{comm} \\ \mathit{new_hist} \ (a, b) \ k &= \mathit{new_int_var_array} \ (a, b) \ \lambda A. \\ &\quad k \ (\mathit{mkcounter} \circ A) \end{aligned}$$

where *new_int_var_array* is a primitive operation for allocating arrays of integer variables. We define two further operations to access the methods of the counter objects:

$$\begin{aligned} \mathit{count} &: \mathbf{histobj} \rightarrow_p (\mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{comm}) \\ \mathit{count} \ h \ i &= (h \ i).\mathit{inc} \\ \mathit{vals} &: \mathbf{histobj} \rightarrow_p \mathbf{histogram} \\ \mathit{vals} \ h \ i &= (h \ i).\mathit{val} \end{aligned}$$

To build a histogram for a list of integers, we can now write:

$$\begin{aligned} & \mathit{build_histogram} : \mathbf{list\ exp[int]} \rightarrow_p \mathbf{exp[int]} \times \mathbf{exp[int]} \rightarrow_p \\ & \quad (\mathbf{histogram} \rightarrow \mathbf{comm}) \rightarrow_p \mathbf{comm} \\ \mathit{build_histogram}\ xs\ (a, b)\ k &= \mathit{new_hist}(a, b)\ \lambda h. \\ & \quad \mathit{mapdo}\ (\mathit{count}\ h)\ xs; \\ & \quad k\ (\mathit{vals}\ h) \end{aligned}$$

These examples must convey to the reader the powerful techniques for building and composing objects made available by the higher-order features of Idealized Algol.

Operational semantics

The operational semantics of the language is defined in two stages [66]. As explained under Discussion above, a program phrase is interpreted by first reducing it as far as necessary and then executing the phrase obtained from reduction. The two stages are formalized as a reduction system and an execution semantics respectively.

The reduction system is expressed by the following reduction rules:

Reductions for higher types:

$$\begin{aligned} \mathit{fst}(P_1, P_2) &\longrightarrow P_1 \\ \mathit{snd}(P_1, P_2) &\longrightarrow P_2 \\ (\lambda x. P) Q &\longrightarrow [Q/x]P \end{aligned}$$

Unfoldings:

$$\mathbf{rec}_\theta P \longrightarrow P (\mathbf{rec}_\theta P)$$

Propagations:

$$\begin{aligned} \mathit{fst}(\mathbf{if}_{\theta \times \theta'}(P_0, P_1, P_2)) &\longrightarrow \mathbf{if}_\theta(P_0, \mathit{fst}(P_1), \mathit{fst}(P_2)) \\ \mathit{snd}(\mathbf{if}_{\theta \times \theta'}(P_0, P_1, P_2)) &\longrightarrow \mathbf{if}_{\theta'}(P_0, \mathit{snd}(P_1), \mathit{snd}(P_2)) \\ \mathbf{if}_{\theta \rightarrow \theta'}(P_0, P_1, P_2) Q &\longrightarrow \mathbf{if}_{\theta'}(P_0, P_1 Q, P_2 Q) \\ \mathbf{if}_{\mathbf{var}[\delta]}(P_0, P_1, P_2) := Q &\longrightarrow \mathbf{if}_{\mathbf{comm}}(P_0, P_1 := Q, P_2 := Q) \end{aligned}$$

The relation \longrightarrow is extended to a reduction relation on phrases (for reduction in all contexts) in the standard fashion. The basic properties of the reduction system may be established as follows:

PROPOSITION 1 (O'HEARN AND TENNENT) *If $\Pi \mid \Gamma \vdash P : \theta$ is a derivable phrase and $P \longrightarrow P'$ then $\Pi \mid \Gamma \vdash P' : \theta$ is a derivable phrase.*

PROPOSITION 2 *The above reduction system is confluent, i.e., if $P \longrightarrow^* P_1$ and $P \longrightarrow^* P_2$ there exists P' such that $P_1 \longrightarrow^* P'$ and $P_2 \longrightarrow^* P'$.*

Proof: Since the left hand sides of the reduction rules have no repeated meta-variables and no critical overlaps, the classical techniques are applicable [8].

■

The second stage of the operational semantics is *execution*. For this stage, we limit attention to expression and command phrases whose free identifiers are all of variable types, i.e., phrases of the form

$$\Delta \mid \Delta' \vdash E : \mathbf{exp}[\delta] \quad \text{and} \quad \Delta \mid \Delta' \vdash C : \mathbf{comm}$$

where $\Delta \mid \Delta'$ is a *variable type context* $x_1 : \mathbf{var}[\delta_1], \dots \mid \dots, x_n : \mathbf{var}[\delta_n]$. We refer to such phrases as *semi-closed* phrases. If $\Delta \mid \Delta'$ is a variable type context, a $\Delta \mid \Delta'$ -state is a finite map of the form $[x_1 \rightarrow i_1, \dots, x_n \rightarrow i_n]$, where each i_k is a data value of type δ_k . Note that the variables mapped by the state are precisely the variables in the type context $\Delta \mid \Delta'$. We use the symbol σ to range over such states. The notation $\sigma[x \rightarrow i]$ means σ with its x component updated to i , and $\sigma_1 \oplus \sigma_2$ denotes the join of two states with disjoint domains.

We define two families of relations:

- $(\sigma, E) \Downarrow_{\Delta, \Delta', \delta} i$, where σ is a $\Delta \mid \Delta'$ -state, E an expression with typing $\Delta \mid \Delta' \vdash E : \mathbf{exp}[\delta]$, and i a data value of type δ .
- $(\sigma, C) \Downarrow_{\Delta, \Delta', \mathbf{comm}} \sigma'$, where σ and σ' are $\Delta \mid \Delta'$ -states and C a command with typing $\Delta \mid \Delta' \vdash C : \mathbf{comm}$.

The first family of relations denotes the evaluation of an expression in a state, and the second family denotes the execution of a command in a state to give a new state. Table 3 gives an inductive definition of the relations for well-typed phrases. For the parallel composition $C_1 \parallel C_2$, execution involves partitioning the state into σ_1 and σ_2 based on the free identifiers occurring in C_1 and C_2 . For the **new** $[\delta]$ combinator, we are assuming that x has been renamed to be distinct from the variable identifiers in σ . The execution involves adding a new component for x in the state and discarding it after the execution of C . This represents the “stack discipline” of local variables in Algol. The new components are initialized to a designated data value $\mathit{init}[\delta]$ upon creation. The last two rules are for phrases derived by the *Contr* rule. Strictly speaking, we should also have rules for phrases derived by *Activate* and *Passify*, but these rules do not affect the execution relations.

PROPOSITION 3 *For all semi-closed expressions E and commands C :*

1. *If $(\sigma, C) \Downarrow \sigma'$ then, for all passive free identifiers x of C , $\sigma(x) = \sigma'(x)$.*
2. *If $(\sigma, E) \Downarrow i$ and $(\sigma, E) \Downarrow i'$ then $i = i'$.*
3. *If $(\sigma, C) \Downarrow \sigma'$ and $(\sigma, C) \Downarrow \sigma''$ then $\sigma' = \sigma''$.*

Proof: By induction on the type derivation of the phrase. ■

A *program* is a closed command phrase. The only observable effect of the program is termination:

Table 3. Execution semantics

$$\begin{array}{c}
\frac{}{(\sigma, 0) \Downarrow 0} \quad \frac{(\sigma, E_1) \Downarrow i_1 \quad (\sigma, E_2) \Downarrow i_2}{(\sigma, E_1 + E_2) \Downarrow i_1 + i_2} \\
\frac{(\sigma, E) \Downarrow tt \quad (\sigma, E_1) \Downarrow i}{(\sigma, \mathbf{if}(E, E_1, E_2)) \Downarrow i} \quad \frac{(\sigma, E) \Downarrow ff \quad (\sigma, E_2) \Downarrow i}{(\sigma, \mathbf{if}(E, E_1, E_2)) \Downarrow i} \\
\frac{}{(\sigma, \mathbf{skip}) \Downarrow \sigma} \quad \frac{(\sigma, C_1) \Downarrow \sigma' \quad (\sigma', C_2) \Downarrow \sigma''}{(\sigma, C_1; C_2) \Downarrow \sigma''} \quad \frac{(\sigma_1, C_1) \Downarrow \sigma'_1 \quad (\sigma_2, C_2) \Downarrow \sigma'_2}{(\sigma_1 \oplus \sigma_2, C_1 \parallel C_2) \Downarrow \sigma'_1 \oplus \sigma'_2} \\
\frac{(\sigma \oplus [x \text{--} \mathit{init}[\delta]], C) \Downarrow \sigma' \oplus [x \text{--} i]}{(\sigma, \mathbf{new}[\delta] x. C) \Downarrow \sigma'} \quad \frac{(\sigma, E) \Downarrow i}{(\sigma, x := E) \Downarrow \sigma[x \text{--} i]} \quad \frac{}{(\sigma, \mathbf{deref} x) \Downarrow \sigma(x)} \\
\frac{(\sigma, E) \Downarrow tt \quad (\sigma, C_1) \Downarrow \sigma'}{(\sigma, \mathbf{if}(E, C_1, C_2)) \Downarrow \sigma'} \quad \frac{(\sigma, E) \Downarrow ff \quad (\sigma, C_2) \Downarrow \sigma'}{(\sigma, \mathbf{if}(E, C_1, C_2)) \Downarrow \sigma'} \\
\frac{(\sigma \oplus [x_1 \text{--} i, x_2 \text{--} i], E) \Downarrow j}{(\sigma \oplus [x \text{--} i], [x/x_1, x/x_2]E) \Downarrow j} \quad \frac{(\sigma \oplus [x_1 \text{--} i, x_2 \text{--} i], C) \Downarrow \sigma' \oplus [x_1 \text{--} j, x_2 \text{--} j]}{(\sigma \oplus [x \text{--} i], [x/x_1, x/x_2]C) \Downarrow \sigma' \oplus [x \text{--} j]}
\end{array}$$

Definition. A closed phrase $\vdash C : \mathbf{comm}$ is said to *terminate* if there is a command $\vdash C' : \mathbf{comm}$ such that $C \longrightarrow^* C'$ and $([], C') \Downarrow []$. Otherwise it is said to *diverge*. Two phrases P and Q are *observationally equivalent* iff, for all program contexts $C[\]$, either $C[P]$ and $C[Q]$ both terminate or they both diverge.

Note that the phrase C' in this definition is some arbitrary reduct of C . It does not have to be any kind of “normal form.” A program C diverges if there is no reduct C' of C for which execution is defined. A typical example is **undef** with the infinite reduction sequence $\mathbf{undef} \longrightarrow \mathbf{undef} \longrightarrow \dots$. A typical example of a terminating program is *while* (**false**, C) which reduces in one step to the executable phrase **if**(**false**, C ; *while* (**false**, C), **skip**).

In practice, one would add constants for input and output and consider equivalence under such observable effects. However, a better focus on the theoretical properties of the language is obtained by considering the coarser equivalence above.

3. Objects

A central tenet of the present semantic model is that states are not simply values, but rather *attributes* of entities that persist in time. To this end, we postulate entities called *objects*.⁶ An object has an internal store (which we think of as a “physical resource”), and a collection of observable operations that potentially read

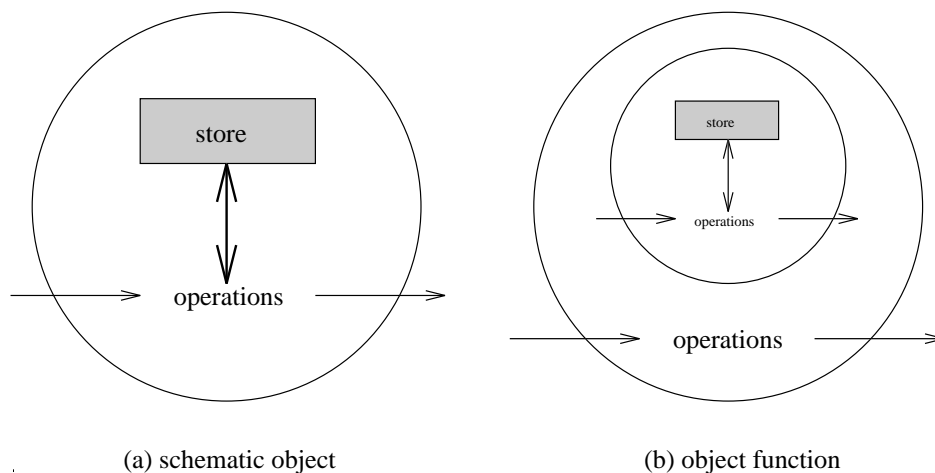


Figure 1.

or alter the store.⁷ See Fig. 1(a) for a schematic picture. The operations have types (**comm**, **exp** $[\delta]$ etc.) which determine how the operations can be used. The types of the operations also determine what kind of semantics is associated with the object.

In devising a mathematical description of objects, we would like to ignore the structure of their stores (because they are purely internal) and focus on their observable operations. To this end, we must have some idea of how an object is to be used and what kind of behavior it supports. We postulate the following hypotheses regarding this behavior.

THEESIS 1 *An object can in general be used only sequentially.*

This is because objects have changeable state in their internal store. Carrying out multiple parallel accesses to an object would cause quite unpredictable effects on its internal state. Thus, sequential access is the only viable option, both theoretically and pragmatically. Note that this does not preclude “concurrent” accesses that are serialized in some fashion at the object boundary (though such concurrency does not arise for the language studied in this paper).

Objects without changeable stores (called “passive” objects) form a special case for which the sequential use restriction can be relaxed. We return to this issue in Sec. 5.

By insisting that objects should be used sequentially, we are requiring that the collection of operations performed on the object be totally ordered, i.e., form a sequence. This gives a local notion of “time” for each object. The operations occurring earlier in the sequence are “before” the operations occurring later.

THESES 2 *The behavior of an object is in general affected by its past history of operations.*

This is the sense in which objects are “dynamic” or “have state”. The past history determines the current state of the object which, in turn, determines the current behavior.

State machines as in classical automata theory have both these properties. We use them as a metaphor for motivating the various issues surrounding objects. Fig. 2 shows the state transition diagrams of several example objects. The nodes in these diagrams should be interpreted as states and arcs as state transitions.

The first object, called a “stepper,” has a single operation that changes the internal state and produces an integer. (It incrementally “steps through” a sequence of integers over a period of time.) We label each transition by the integer that is produced. In any given run of a program, a single sequence of integers can be extracted from the object. Similarly, for any object, a single sequence of observations can be made in any given run of a program. We will call such sequences of observations *traces*, in analogy with Hoare’s terminology for processes [28], and the set of all traces that can be observed from the object its *trace set*. Note that the trace set is a state-free description of the observable behavior of an object, much like the notion of a “language” of an automaton. The trace set of the stepper object consists of sequences:

$$\langle \rangle, \langle 0 \rangle, \langle 0, 1 \rangle, \langle 0, 1, 2 \rangle, \dots$$

The trace set is *prefix-closed* because every state of the object is (implicitly) regarded as a terminal state.

Fig. 2(b) shows the behavior of the counter object discussed in Sec. 2. Invoking its *val* operation returns the current value of the counter and makes no change to the state. Invoking the *inc* operation changes its state and returns a dummy completion signal that we write as $*$. So, the trace set of the counter object is:⁸

$$\begin{aligned} &\langle \rangle, \langle \text{val}.0 \rangle, \langle \text{inc}.* \rangle \\ &\langle \text{val}.0, \text{val}.0 \rangle, \langle \text{val}.0, \text{inc}.* \rangle, \langle \text{inc}.*, \text{val}.1 \rangle, \langle \text{inc}.*, \text{inc}.* \rangle, \\ &\dots \end{aligned}$$

assuming the counter starts with an initial value of 0.

Our thesis is that all types of Algol similarly give rise to objects. Figures 2(c-e) show canonical example objects corresponding to the primitive types of Algol. A *command object* is an object with a single command as its operation. Invoking it potentially changes the internal state and returns a completion signal. For example, defining:

$$\begin{aligned} \text{inc} &: \mathbf{var}[\mathbf{int}] \rightarrow_p \mathbf{comm} \\ \text{inc}(x) &= (x := x + 1) \end{aligned}$$

we can construct a command object from a variable object. The procedures p in equivalences (1) and (2) of Sec. 1 receive such command objects as their arguments.

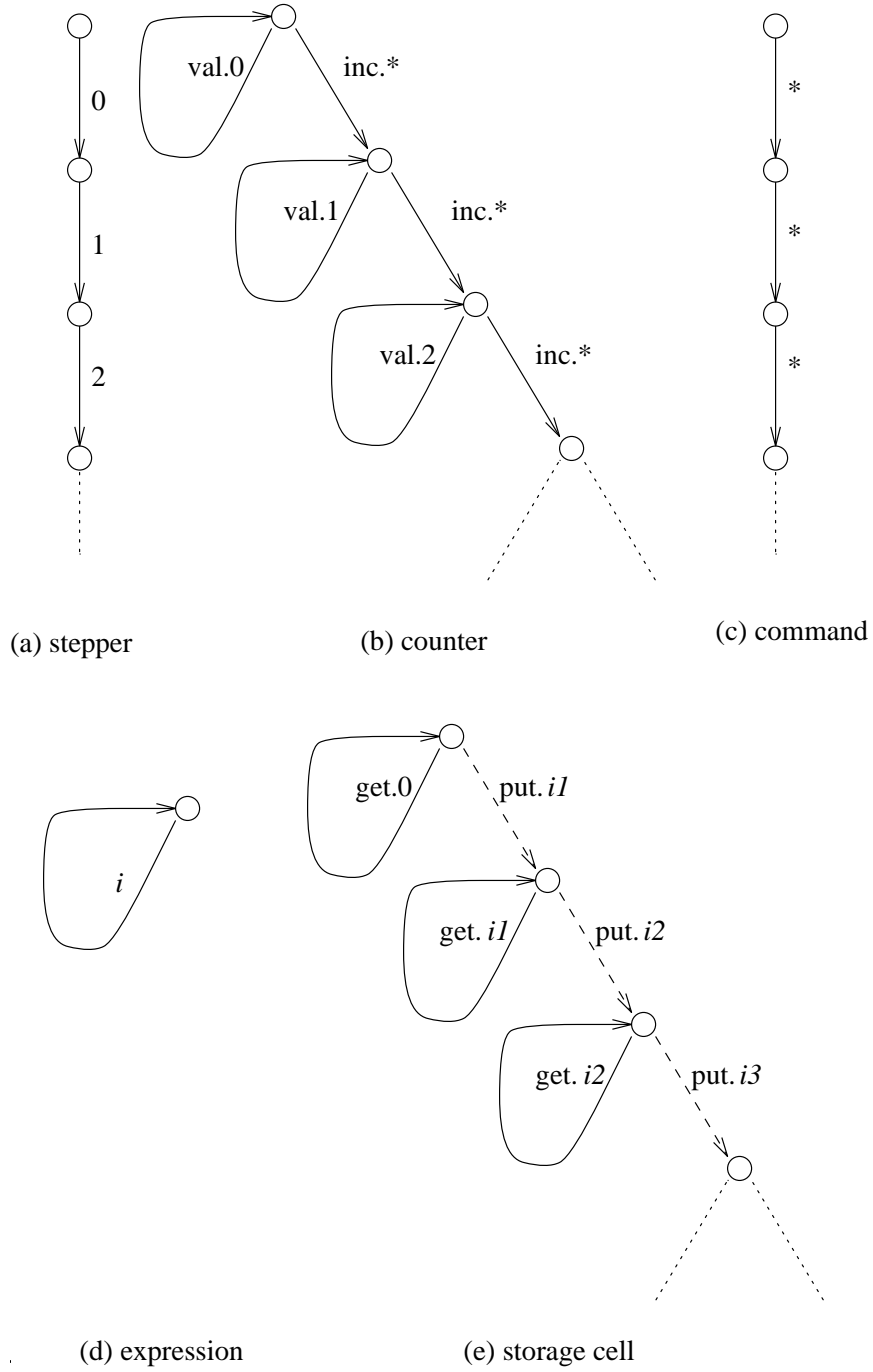


Figure 2. Object behaviors

The behavior shown in Fig. 2(c) is what is observable by such procedures. The effect of the command on the variable x will be represented in the meaning of the function inc , explained below. Since a command object has changeable internal state, it need not produce “*” in every state. For instance, the phrase

$$x : \mathbf{var}[\delta] \vdash \mathbf{if } x < n \mathbf{ then } x := x + 1 \mathbf{ else } \mathbf{undef} : \mathbf{comm}$$

produces a command object that eventually reaches a diverging state.

The *expression object* in Fig. 2(d) returns a value for each invocation of its operation and makes no change to the state. This corresponds to the fact that expressions in Idealized Algol do not cause side effects. Objects that do not cause state changes will be called passive objects.

Note that projecting the *val* and *inc* operations of a counter object gives an expression object and a command object respectively. This is the sense in which a counter is a “pair” of an expression and a command. In general, “pairing” involves pairing of methods, not pairing of objects.

Finally, a canonical variable object called a *storage cell* has the behavior shown in Fig. 2(e). It has an operation *get* to retrieve the value stored in the cell and an operation *put* to store a new value. The dashed arcs emanating from each state denote potentially infinite families of arcs, one for each data value i_k . Notice that each *get* operation retrieves the value previously stored in the cell. This explanation shows that storage cells can be interpreted as “pairs” (in the same sense as counters are “pairs”). The *get* components are expression objects while the *put* components are called “acceptors” [66]. The two components correspond to what are often called the *r*-value and the *l*-value of a variable.

A *function* on objects allows us to construct one object from another object as shown in Fig. 1(b). An example is the function *mkcounter* which constructs a counter object from a variable object. The effect of such a construction is to simulate every operation on the outer object by a sequence of operations on the inner object and translate their results back to a result of the outer object. Such simulations can be described compactly by maps that we call *patterns*. The pattern for the *mkcounter* function has input-output pairs of the form

$$\begin{aligned} \langle \mathit{get}.i \rangle &\mapsto \mathit{val}.i \\ \langle \mathit{get}.i, \mathit{put}.(i + 1) \rangle &\mapsto \mathit{inc}.* \end{aligned}$$

for all integer values i . This shows that the *val* operation of the counter object is simulated by a *get* operation on the variable object and the *inc* operation of the counter object is simulated by an appropriate *get-put* sequence on the variable object.⁹

Note that, in such a simulation, the internal store of the original object becomes the internal store of the new object. Thus, the nested object picture in Fig. 1(b) is quite appropriate. At the same time, the function carrying out the simulation has no direct access to the internal state of the original object. It can only affect state changes via the *operations* of the object.

This fact allows us to ignore the internal states of objects and work entirely at the level of operations and behaviors. It leads to a domain-theoretic model of interference-controlled Algol presented in the next two sections. Yet, imperative programs are often thought of as acting on state sets. In the remainder of this section, we indicate how object behaviors relate to an explicit state point of view.

Automata and simulations

We can illustrate the structure of objects for simple cases using the classical notions of automata. An *automaton* for an instruction set Σ (or a Σ -*automaton*) is a pair $(Q, \alpha : Q \times \Sigma \rightarrow Q)$ where Q is a set and α is a partial function (written as an infix operator). It is standard practice to extend the function α to a function $\bar{\alpha} : Q \times \Sigma^* \rightarrow Q$ by

$$q \bar{\alpha} \langle a_1, \dots, a_n \rangle = (\dots (q \alpha a_1) \dots \alpha a_n)$$

We call Q the set of *states* of the automaton and α its *transition function*. By picking appropriate instruction sets, we can define automata for various types.

For example, taking $\Sigma = \{*\}$, we obtain automata that correspond to command objects. They have a single instruction $*$ for running the command once and instruction sequences $\langle * \rangle^n$ achieve the effect of running the command n times.

The *behavior* (or the *language*) of an automaton (Q, α) at a state $q \in Q$ is defined by

$$L_\alpha(q) = \{ x \in \Sigma^* : q \bar{\alpha} x \text{ is defined} \}$$

It is easy to see that $L_\alpha(q)$ is always a prefix-closed set. Moreover, given any prefix-closed set $X \subseteq \Sigma^*$, we can define an automaton (Q, α) such that its language at a designated state $q \in Q$ is X . A simple construction is to pick $Q = X$ and define α by $\alpha : (x, a) \mapsto x \cdot \langle a \rangle$ whenever $x \cdot \langle a \rangle \in X$. The start state q is the empty sequence $\langle \rangle$. One can also construct a minimal automaton by merging all equivalent states in (Q, α) . Thus, up to behavioral equivalence, automata with designated start states are the same as prefix-closed subsets of Σ^* . We make much use of this fact in this paper.

Let Σ and Γ be instruction sets. Consider a function f from Σ -automata to Γ -automata that represents a simulation. Our intuitions about simulations suggest that the instructions of Γ must be simulated by sequences of instructions of Σ . In other words, f is uniquely determined by a *pattern map* $h : \Gamma \rightarrow \Sigma^*$, a partial function. Note the reversal of the direction. The function f on automata is then given by:

$$f(Q, \alpha : Q \times \Sigma \rightarrow Q) = (Q, \beta : Q \times \Gamma \rightarrow Q) \text{ where} \quad (3)$$

$$q \beta b = q \bar{\alpha} h(b)$$

One sees that such functions f are intuitively independent of (or uniform in) the underlying state sets of the automata. More precisely, one can state the following correspondence (pointed out to us by John Gray):

PROPOSITION 4 *Functions $f : \Sigma\text{-Aut} \rightarrow \Gamma\text{-Aut}$ that preserve the underlying state sets and state mappings (homomorphisms of automata) are one-to-one with partial functions $h : \Gamma \rightarrow \Sigma^*$.*

Proof: Suppose $h : \Gamma \rightarrow \Sigma^*$ is a partial function. Consider the function f given by (3). It evidently preserves state sets. Let $t : (Q, \alpha) \rightarrow (Q', \alpha')$ be a state mapping of Σ -automata, i.e., a function $t : Q \rightarrow Q'$ satisfying $t(q) \alpha' a = t(q \alpha a)$. Clearly, $t(q) \bar{\alpha}' x = t(q \bar{\alpha} x)$ for all $x \in \Sigma^*$. Denoting $f(Q, \alpha)$ by (Q, β) and $f(Q', \alpha')$ by (Q', β') , it follows that $t(q) \beta' b = t(q \beta b)$ for all $b \in \Gamma$. So, t is a state mapping of Γ -automata.

Conversely, let $f : \Sigma\text{-Aut} \rightarrow \Gamma\text{-Aut}$ be a function preserving state sets and state mappings. There exists a “free” Σ -automaton (Σ^*, α_0) where $x \alpha_0 a = x \cdot \langle a \rangle$. Let its image under f be (Σ^*, β_0) . We can define the pattern map $h : \Gamma \rightarrow \Sigma^*$ by $h(b) = \langle \rangle \beta_0 b$. Given any Σ -automaton (Q, α) and state $q \in Q$, there exists a state mapping $\tilde{q} : (\Sigma^*, \alpha_0) \rightarrow (Q, \alpha)$ given by $\tilde{q}(x) = q \alpha x$. From the fact that all such state mappings are preserved, one can verify that f satisfies equation (3). ■

Simulations of this form correspond closely to phrases of interference-controlled Algol. For example, consider command objects treated as $\{*\}$ -automata. A function from command objects to command objects is determined by a map $h : \{*\} \rightarrow \{*\}^*$. What maps are there? There is the undefined map and, for every natural number n , there is a map $h_n(*) = \langle * \rangle^n$. Each of these maps determines a function at the level of automata. The function corresponding to h_n is:

$$f_n(Q, \alpha : Q \times \{*\} \rightarrow Q) = (Q, \beta : Q \times \{*\} \rightarrow Q) \text{ where} \\ q \beta * = q \bar{\alpha} \langle * \rangle^n$$

All such functions are realizable in interference-controlled Algol: the function f_n corresponds to the function phrase *times n* mentioned in Sec. 2. So, already at this elementary level, the object-based viewpoint provides a solution for one of the key problems in the semantics of state.

From automata to objects

The simple notions of automata outlined above are not sufficient for modelling objects in general. While the instructions of automata are purely inputs, the operations of objects have input as well as output information. For example, the operation of a command object has essentially input information as explained above. In contrast, the operation of an expression object has essentially output information. Its transition function is of the form $\alpha : Q \rightarrow Z \times Q$ (where Z is the set of integers). Objects with higher type operations will have more complex breakdown of input and output information. We would like to cover all cases uniformly.

For this purpose, we make use of a standard fact. A (partial) function $f : A \rightarrow B$ is nothing but a special kind of a relation $f \subseteq A \times B$. What distinguishes a function

from general relations is that any given input determines at most one output in the relation, i.e., for all $(x, y), (x', y') \in f$, $x = x' \implies y = y'$. Let us say that two input-output pairs (x, y) and (x', y') are *consistent*, and write $(x, y) \circ (x', y')$, when this property holds. Then a function $f : A \rightarrow B$ is nothing but a relation $f \subseteq A \times B$ whose input-output pairs are pairwise consistent. The causality information missing in relations can be brought back by defining a separate consistency relation in this fashion. The advantage is that complex causality relationships can be expressed using consistency relations.

So, in a first attempt, we let each type θ determine a set of values Σ and a binary consistency relation \circ on $Q \times \Sigma \times Q$. A *transition map* is a relation $\alpha \subseteq Q \times \Sigma \times Q$ such that the tuples of α are pairwise consistent. For command objects, we let $\Sigma = \{*\}$ and define the consistency relation for transition maps to be:

$$(q_1, *, q'_1) \circ (q_2, *, q'_2) \iff (q_1 = q_2 \implies q'_1 = q'_2)$$

For integer expression objects, let $\Sigma = Z$ and define the consistency relation for transition maps to be:

$$(q_1, a_1, q'_1) \circ (q_2, a_2, q'_2) \iff (q_1 = q_2 \implies a_1 = a_2 \wedge q'_1 = q'_2)$$

For integer variable objects, Σ is defined to be the disjoint union of sets

$$Z + Z = \{ \text{get}.a : a \in Z \} \cup \{ \text{put}.b : b \in Z \}$$

The consistency relation for transition maps is:

$$\begin{aligned} (q_1, l_1.a_1, q'_1) \circ (q_2, l_2.a_2, q'_2) \iff \\ (l_1 = l_2 = \text{get} \implies q_1 = q_2 \implies a_1 = a_2 \wedge q'_1 = q'_2) \wedge \\ (l_1 = l_2 = \text{put} \implies q_1 = q_2 \wedge a_1 = a_2 \implies q'_1 = q'_2) \end{aligned}$$

Note how the consistency relation handles the fact that the *get* operations have output information and the *put* operations have input information. More generally, suppose that types θ_1 and θ_2 have sets of values Σ_1 and Σ_2 and transition consistency relations \circ_1 and \circ_2 respectively. Then, for the product type $\theta_1 \times \theta_2$, we let Σ be the disjoint union $\Sigma_1 + \Sigma_2 = \{ 1.a : a \in \Sigma_1 \} \cup \{ 2.b : b \in \Sigma_2 \}$. The consistency relation for transition maps is

$$(q_1, i.a_1, q'_1) \circ (q_2, j.a_2, q'_2) \iff (i = j \implies (q_1, a_1, q'_1) \circ_i (q_2, a_2, q'_2))$$

In all these cases, the consistency relation for transition maps can be given uniformly, provided we admit a consistency relation for values themselves. This leads to a second attempt where we associate with each type θ , a set of values Σ and a binary consistency relation \circ on Σ itself. The consistency relation for transition maps is then defined uniformly by:

$$\begin{aligned} (q_1, a_1, q'_1) \circ (q_2, a_2, q'_2) \iff (q_1 = q_2 \implies a_1 \circ a_2) \wedge \\ (q_1 = q_2 \wedge a_1 = a_2 \implies q'_1 = q'_2) \end{aligned} \tag{4}$$

If we assign the following sets of values and consistency relations for the types mentioned above:

$$\begin{array}{llll}
\mathbf{comm} & \Sigma = \{*\} & \circ & = \text{identity relation} \\
\mathbf{exp[int]} & \Sigma = Z & \circ & = \text{identity relation} \\
\mathbf{var[int]} & \Sigma = Z + Z & l.a \circ l'.b & \iff l = l' = \text{get} \implies a = b \\
\theta_1 \times \theta_2 & \Sigma = \Sigma_1 + \Sigma_2 & i.a \circ j.b & \iff i = j \implies a \circ_i b
\end{array}$$

it can be verified that definition (4) gives the same consistency relation for transition maps as defined earlier. The consistency relation assigned to value sets may be intuitively understood as follows: $a_1 \circ a_2$ means that either a_1 and a_2 have differing input information or they have the same output information. This motivates the condition in (4) that, whenever a_1 and a_2 are observed in the same state of an object, they must be consistent. Conversely, define the *inconsistency* relation \succsim on Σ by $a_1 \succsim a_2$ iff $a_1 = a_2 \vee \neg(a_1 \circ a_2)$. Now, $a_1 \succsim a_2$ signifies that a_1 and a_2 have the same input information.

Next, we extend the consistency relations to sequences Σ^* . The extension of transition maps to sequences is the natural one: $(q, \langle a_1, \dots, a_n \rangle, q') \in \bar{\alpha}$ iff there exist states q_0, \dots, q_n such that $q = q_0, q_n = q'$ and

$$(q_0, a_1, q_1), \dots, (q_{n-1}, a_n, q_n) \in \alpha$$

The relation $\bar{\alpha}$ must be a transition map consistent in the sense of (4). This forces a consistency relation on sequences in Σ^* , viz., $\langle a_1, \dots, a_n \rangle \circ \langle a'_1, \dots, a'_m \rangle$ iff

$$\langle a_1, \dots, a_{i-1} \rangle = \langle a'_1, \dots, a'_{i-1} \rangle \implies a_i \circ a'_i \quad \text{for all } i = 1, \dots, \min(n, m) \quad (5)$$

Note Thesis 2 at play here. The past events in a trace a_1, \dots, a_{i-1} determine the future event a_i (modulo its own input part). It is easy to see that, whenever, $\alpha \subseteq Q \times \Sigma \times Q$ is a transition map, its extension $\bar{\alpha} \subseteq Q \times \Sigma^* \times Q$ is also a transition map.

As an example, a storage cell object with the behavior shown in Fig. 2(e) can be defined by taking the state set to be the set of integers and the transition map given by:

$$\begin{array}{ll}
(i, \text{get}.j, i') \in \alpha & \iff i = j \wedge i = i' \\
(i, \text{put}.j, i') \in \alpha & \iff i' = j
\end{array} \quad (6)$$

It may be verified that this is a proper transition map satisfying the consistency condition (4).

The *trace set* of an object (Q, α) at a state $q \in Q$ is defined by

$$L_\alpha(q) = \{x \in \Sigma^* : \exists q' \in Q. (q, x, q') \in \bar{\alpha}\}$$

This is a pairwise consistent set by the above consistency relation for Σ^* .

Finally, we consider “functions” on objects. A function of type $\theta_1 \rightarrow \theta_2$ must map objects with transition maps $\alpha_1 \subseteq Q \times \Sigma_1 \times Q$ to objects with transition

maps $\alpha_2 \subseteq Q \times \Sigma_2 \times Q$. As argued previously, any such “function” must be determined by a pattern map $h \subseteq \Sigma_1^* \times \Sigma_2$. The map records the sequence of input instructions (in Σ_1^*) required to simulate an output instruction (in Σ_2). To see what kind of a consistency relation must be satisfied by h , let $(x, b) \in h$. First, the input information of b must uniquely determine the input information of x . So, if $(x', b') \in h$ is another pair, we must have:

$$b \succ b' \implies x \succ x' \tag{7}$$

Second, for any given input information of b , the output information of x must uniquely determine the output information of b , i.e.,

$$b \succ b' \wedge x = x' \implies b = b' \tag{8}$$

It is often more convenient to express these conditions in the forward direction: two pairs $(x, b), (x', b') \in h$ are consistent iff

$$(x \circ x' \implies b \circ b') \wedge (x \circ x' \wedge b = b' \implies x = x') \tag{9}$$

With some work, it can be seen that this definition is equivalent to the original conditions (7) and (8). Maps of this kind are called *linear maps*. We examine them in more detail in Sec. 4.2.

As in the case of automata, a linear pattern map h uniquely determines a function at the level of objects:

$$f(Q, \alpha_1 \subseteq Q \times \Sigma_1 \times Q) = (Q, \alpha_2 \subseteq Q \times \Sigma_2 \times Q) \text{ where} \tag{10} \\ (q, b, q') \in \alpha_2 \iff \exists x \in \Sigma_1^*. (x, b) \in h \wedge (q, x, q') \in \bar{\alpha}_1$$

Note that x in the above existential is unique.

An *object* of a function type $\theta_1 \rightarrow \theta_2$ is similar to a function. Its value set is $\Sigma_F = \Sigma_1^* \times \Sigma_2$ and its consistency relation is defined by (9). However, being an object, it can have its own internal store and use it to control the simulation of operations in a *history-sensitive* fashion. So, its application to an argument object of type θ_1 yields an object whose behavior is controlled by both the store of the function object and the store of the argument object. So, we can view a function object $(Q_F, \alpha_F \subseteq Q_F \times \Sigma_F \times Q_F)$ as determining a function from objects (Q_1, α_1) to objects $(Q_F \times Q_1, \alpha_2)$ with transition maps $\alpha_2 \subseteq (Q_F \times Q_1) \times \Sigma_2 \times (Q_F \times Q_1)$ given by

$$((q, q_1), b, (q', q'_1)) \in \alpha_2 \iff \exists x \in \Sigma_1^*. (q, (x, b), q') \in \alpha_F \wedge (q_1, x, q'_1) \in \bar{\alpha}_1$$

Thus, function-type values expressible in imperative languages can have a richer behavior than functions themselves. One must take care to keep the two notions separate.

Reynolds’s passive function type $\theta_1 \rightarrow_p \theta_2$ brings back a correspondence. Since terms of type $\theta_1 \rightarrow_p \theta_2$ cannot have free active identifiers, the objects they denote do not have changeable stores. Such (passive) objects precisely correspond to functions $\theta_1 \rightarrow \theta_2$.

In the next two sections, we formalize these concepts using the semantic framework of coherent spaces.

4. A Domain-Theoretic Model of Objects

In this section, we present a domain-theoretic model of object behaviors. The central fact is that object behaviors form a certain kind of domains, and “functions” on objects can be characterized as a class of functions between such domains. While the internal structure of objects, described in the previous section, can be used for intuitive understanding, the formal analysis can be carried out entirely in terms of behaviors.

The domains we use for this purpose are Girard’s *coherent spaces* [17]. The basic source on coherent spaces is [19], Chapters 8 and 12, while Zhang [85] discusses the connections with other kinds of domains used in the literature.

A *coherent space* A is a domain of *sets*, ordered by the subset order, such that A is

- *down-closed*: if $x \in A$ and $y \subseteq x$ then $y \in A$, and
- *coherent*: if $X \subseteq A$ and $\forall x, y \in X. x \cup y \in A$ then $\bigcup X \in A$.

Note that down-closure implies that lub’s and glb’s are given by union and intersection respectively. The bottom element is \emptyset . Experts in domain theory might note that, up to order-isomorphism, coherent spaces are precisely dI-domains that are coherent (all pairwise-consistent sets have lub’s) and atomic (the elements covering \perp form a sub-basis). A *linear function* $f : A \rightarrow_L B$ between coherent spaces is a continuous function such that

1. $x \cup y \in A \implies f(x \cap y) = f(x) \cap f(y)$, and
2. $X \subseteq A \wedge (\forall x, y \in X. x \cup y \in A) \implies f(\bigcup X) = \bigcup f(X)$.

In other words, linear functions preserve glb’s of consistent pairs (*stability*) and lub’s of pairwise consistent sets (*linearity*). Coherent spaces and linear functions form a closed (but not cartesian-closed) category.

In practice, we work with a representation of coherent spaces called a “web.” Suppose A is a coherent space. Define:

- $|A| = \bigcup A$, the set of elements of elements of A . These elements are called the “atoms” or “tokens” of A .
- A binary relation of “consistency” $\circ_A \subseteq |A| \times |A|$, defined by $a \circ_A a' \iff \{a, a'\} \in A$. Note that \circ_A is reflexive and symmetric.

We can recover the domain A from the web. The elements of A are all pairwise consistent sets (also called “coherent sets”) over $|A|$. So, webs (countable sets with reflexive-symmetric relations) form canonical representations for coherent spaces, and we use them as such.

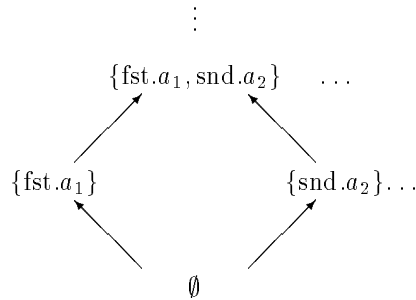
We argued in Sec. 3 that the types of interference-controlled Algol should determine sets with binary consistency relations. We now see that such sets with consistency relations represent coherent spaces — a well-known class of domains.

Table 4. Common coherent spaces

$ bool = \{tt, ff\}$	$\circ_{bool} =$ the identity relation
$ int = \{0, 1, 2, \dots\}$	$\circ_{int} =$ the identity relation
$ A_1 \times A_2 = A_1 + A_2 = \{fst.a : a \in A_1 \} \cup \{snd.a : a \in A_2 \}$	$l.a \circ_{l'.a'} \iff (l = l' \implies a \circ_{A_i} a')$
$ A \otimes B = A \times B $	$(a, b) \circ (a', b') \iff a \circ_A a' \wedge b \circ_B b'$
$\top = \emptyset$	$\circ_{\top} = \emptyset$
$\mathbf{1} = \{*\}$	$\circ_{\mathbf{1}} =$ the identity relation
$ A \multimap B = A \times B = \{a \mapsto b : a \in A , b \in B \}$	$(a, b) \circ (a', b') \iff (a \circ_A a' \implies b \circ_B b') \wedge (a \circ_A a' \wedge b = b' \implies a = a')$

The tokens of a coherent space should be regarded as atomic pieces of information about its elements which can be extracted in a single “use” of an element. The consistency relation \circ_A states whether two such pieces of information can coexist in an element.

Table 4 gives the definitions of coherent spaces we use in this paper (via their web representations). The spaces *bool* and *int* are flat domains. The domain $A_1 \times A_2$ is the product of A_1 and A_2 . A token of $A_1 \times A_2$ carries an atomic piece of information about either an element of A_1 or an element of A_2 . So, $|A_1 \times A_2|$ is the disjoint union $|A_1| + |A_2|$. The token *fst.a* corresponds to the element conventionally written as (a, \perp) while *snd.a* corresponds to (\perp, a) . Since the two kinds of tokens are considered consistent, the domain $A_1 \times A_2$ has the expected structure:



When we use labelled products, such as $[l_1 : A_1 \times l_2 : A_2]$, we use the labels l_1 and l_2 as the tags for tokens instead of *fst* and *snd*.

The domain $A \otimes B$ is called the *tensor product*. It is an “eager” kind of product in the sense that each token of $A \otimes B$ contains information about both an A component and a B component.¹⁰ Sometimes, we use labelled tensor products such as $[l_1 : A_1 \otimes \dots \otimes l_n : A_n]$. In that case, the tokens of the coherent space are treated as finite maps $[l_1 \mapsto a_1, \dots, l_n \mapsto a_n]$ where each $a_i \in |A_i|$. The domain \top has a single undefined element and the domain $\mathbf{1}$ is a two point lattice. It is easily verified

that \top and $\mathbf{1}$ are the units of \times and \otimes respectively: $A \times \top \cong A \cong \top \times A$, $A \otimes \mathbf{1} \cong A \cong \mathbf{1} \otimes A$.

Under the action of linear functions, the product type behaves as a “choice” rather than a conventional product. If $f : A \times B \rightarrow_L C$ is a linear function, then the action of f on $\{\text{fst}.a, \text{snd}.b\}$ is determined by its action on $\{\text{fst}.a\}$ and $\{\text{snd}.b\}$. In other words, every atomic piece of information in the output of f is derived from either the A or the B component of the pair, but not both. For instance, there is no linear function of type $\text{int} \times \text{int} \rightarrow_L \text{int}$ that represents addition. On the other hand, the tensor product $A \otimes B$ is defined so as to make information of both the components available for a linear function. Note, for example, that the addition of integers is representable by a linear function of type $\text{int} \otimes \text{int} \rightarrow_L \text{int}$ (cf. Table 5.)

To explain $A \multimap B$ (called the “linear function space”), we need a further analysis of linear functions. Consider a linear function $f : A \rightarrow_L B$ between coherent spaces. Suppose $b \in f(x)$. Since f preserves (consistent) glb’s, there exists a least (finite) element $x_0 \subseteq x$ such that $b \in f(x_0)$. This is the import of the preservation of consistent glb’s: every atomic piece of information in $f(x)$ “comes from” a unique approximation x_0 of x . Since f also preserves lub’s, x_0 must be a singleton $\{a\}$. This then is the significance of preservation of lub’s. Every atomic piece of information in $f(x)$ comes from a unique atomic piece of information in x . In this fashion, linear functions capture the intuition of functions that “use their argument precisely once.” Define a relation $\mu f \subseteq |A| \times |B|$, called the *linear map* of f , by

$$\mu f = \{ a \mapsto b : b \in f(\{a\}) \}$$

From the linear map, we can recover the function f by:

$$f(x) = \{ b : \exists a \in x. a \mapsto b \in \mu f \}$$

This correspondence is a bijection.

PROPOSITION 5 (GIRARD) *Let $f : A \rightarrow_L B$ be a linear function between coherent spaces and μf its linear map. Then, for all $a_1 \mapsto b_1, a_2 \mapsto b_2 \in \mu f$,*

1. $a_1 \circ_A a_2 \implies b_1 \circ_B b_2$, and
2. $a_1 \circ_A a_2 \wedge b_1 = b_2 \implies a_1 = a_2$.

Conversely, any relation $F \subseteq |A| \times |B|$ that satisfies these properties is the linear map of a linear function.

Property 1 is an obvious consequence of monotonicity, while property 2 follows from stability. Recall that these two conditions were already seen in Sec. 3 as condition (9). They capture the bidirectional flow of information involved in linear functions. Condition 1 captures the forward causality that inputs “give rise” to outputs. Condition 2 captures the intuition that individual “demands” for outputs give rise to unique demands for inputs. From now on, we will call any relation satisfying these conditions a *linear map* and regard linear maps as canonical representations for linear functions.

Table 5. Examples of linear functions

id_A	$: A \rightarrow_L A$	$= \{a \mapsto a : a \in A \}$
$g \circ f$	$: A \rightarrow_L C$	$= \{a \mapsto c : \exists b. a \mapsto b \in f \wedge b \mapsto c \in g\}$
$+$	$: int \otimes int \rightarrow_L int$	$= \{(i, j) \mapsto i + j : i, j \in int \}$
$fst_{A,B}$	$: A \times B \rightarrow_L A$	$= \{fst.a \mapsto a : a \in A \}$
$snd_{A,B}$	$: A \times B \rightarrow_L B$	$= \{snd.b \mapsto b : b \in B \}$
$\langle f, g \rangle$	$: C \rightarrow_L A \times B$	$= \{c \mapsto fst.a : c \mapsto a \in f\} \cup \{c \mapsto snd.b : c \mapsto b \in g\}$
$!_A$	$: A \rightarrow \top$	$= \emptyset$

Table 5 gives examples of linear maps. The reader would find it instructive to verify that they correspond to the linear functions they are meant to represent. An important aspect of the composition operator is that the token b in the existential is unique. (Suppose b' is another token such that $a \mapsto b' \in f$ and $b' \mapsto c \in g$. Since $a \circlearrowleft_A a$, we have $b \circlearrowleft_B b'$. But, then, $b \mapsto c, b' \mapsto c \in g$ implies that $b = b'$.) Most existentials that arise with linear functions are similarly forced to be unique.

Returning to Table 4, taking the conditions of Proposition 5 as the consistency relation yields a coherent space $A \multimap B$, called the linear function space of A and B . Note that the pairwise consistent sets of $A \multimap B$ are precisely linear maps. The partial order of $A \multimap B$ is the subset order on linear maps. Given that linear maps are one-to-one with linear functions, there is a corresponding partial order on linear functions which may be expressed as follows:

$$f \sqsubseteq g \iff \forall x, y \in A. x \subseteq y \implies f(x) = f(y) \cap g(x)$$

This is called the stable order or Berry order [10]. It is notably different from the pointwise order commonly used with continuous functions on Scott domains.¹¹

There is an isomorphism of linear functions:

$$A \otimes B \rightarrow_L C \cong A \rightarrow_L (B \multimap C) \tag{11}$$

which establishes that coherent spaces and linear maps form a (symmetric monoidal) closed category. The maps in the two directions give the standard combinators for higher-order functions. For the left-to-right direction, if $f : A \otimes B \rightarrow_L C$ is a linear map, the corresponding map ($\text{curry } f$) : $A \rightarrow_L (B \multimap C)$ is defined by

$$\text{curry } f = \{a \mapsto (b \mapsto c) : (a, b) \mapsto c \in f\}$$

It can be verified that this is a proper linear map and that curry is invertible.

4.1. Object Spaces

Recall that an event of an object trace must correspond to the information obtained via a *single use* of the object. In a coherent space, the information that can be

Table 6. Coherent spaces for Algol types

$ exp[\delta] = \delta $	$\circ_{exp[\delta]} =$ the identity relation
$ comm = \{*\}$	$\circ_{comm} =$ the identity relation
$ var[\delta] = \delta + \delta = \{\text{get}.i : i \in \delta \} \cup \{\text{put}.i : i \in \delta \}$	$l.i \circ l'.i' \iff (l = l' = \text{get} \implies i = i')$

extracted in a single use is represented by the tokens. So, the trace of an object with observable operations of type A must be a sequence of tokens of A . This formalizes Thesis 1. For example, assuming that **exp[int]** is modelled by the domain int and **comm** by the domain $\mathbf{1}$, we have a domain $counter = [\text{val} : int \times \text{inc} : \mathbf{1}]$ with tokens in $|counter| = \{\text{val}.i : i \in |int|\} \cup \{\text{inc}.*\}$. Each event of a counter trace in Fig. 2(b) is a token of this domain.

We define a domain construction for object behaviors as follows:

Definition. If A is a coherent space, the (*free*) *object space* of A , denoted $\dagger A$, is a coherent space that has

- tokens $|\dagger A| = |A|^*$, and
- consistency relation $\langle a_1, \dots, a_n \rangle \circ_{\dagger A} \langle a'_1, \dots, a'_m \rangle$ iff

$$\langle a_1, \dots, a_{i-1} \rangle = \langle a'_1, \dots, a'_{i-1} \rangle \implies a_i \circ_A a'_i \quad \text{for } i = 1, \dots, \min(n, m)$$

(A general notion of object spaces is given in Appendix A.)

The consistency relation $\circ_{\dagger A}$ is the same as (5) from Sec. 3. Essentially, two sequences are consistent if, at the first point of difference between them (if any), they differ consistently. If they do not differ at any position, i.e., one of them is a prefix of the other, they are vacuously consistent.

For every element $x \in A$, there is a corresponding element $x^* \in \dagger A$ (where x^* denotes the set of all sequences over x). We call such elements *regular elements*. They correspond to behaviors of objects that have no (observable) mutable state. The non-empty prefix-closed elements are trace sets. They indicate the behavior of objects starting from a particular state. We call them *active elements*. The remaining elements model the behavior of objects with designated start and final states.

The elements of $\dagger A$ model object behaviors starting from a particular *state*. So, in a sense, they are also modelling states. The empty trace set models the undefined state. If x is a trace set and s is a trace, then $x/s = \{t : s \cdot t \in x\}$ models the state obtained by carrying out the sequence of operations s on an object in state x .

Table 6 shows the definitions of coherent spaces for interpreting the primitive types of interference-controlled Algol. The domain $exp[\delta]$ (for a data type δ) is defined to be the same as the domain for δ . Only flat domains are allowed for the

interpretation of data types. The elements of $\dagger exp[\delta]$ model expression objects with possible side effects, e.g., the stepper object of Fig. 2(a). This will be refined in Sec. 5 for modelling side effect-free expressions. The domain $comm$ has a single token “*” which is thought of as the termination signal of a command. The domain $var[\delta]$ models the operations of variable objects. Note that two *put* tokens are always consistent (signifying that any value can be stored in a given state). A *get* token is always consistent with a *put* token. On the other hand, two *get* tokens are consistent only if the value obtained is the same in both cases. (The transitions of storage cell shown in Fig. 2(e) are pairwise-consistent in this sense.) These domains correspond to the *operations* of objects. The domain of object *behaviors* is obtained by applying the \dagger operator to them.

The elements of the domain $\dagger var[\delta]$ model not only what we normally think of as storage variables (or storage cells), but arbitrary objects with a *get* and a *put* operation (where *get* returns a data value and *put* accepts a data value). For example, we can have a trace of the form $\langle put.1, get.0 \rangle$ which, somewhat counter-intuitively, returns a value different from the one previously stored. Variable objects with such behavior are, however, expressible in interference-controlled Algol. For example, assuming free identifiers $x : \mathbf{var}[\mathbf{int}]$ and $y : \mathbf{var}[\mathbf{int}]$, the phrase

```

let  $v = (\mathbf{if} \ x = 0 \ \mathbf{then} \ x \ \mathbf{else} \ y)$ 
in  $v := 1; \mathbf{print}(v)$ 

```

prints 0 rather than 1 (assuming both x and y have the initial value 0.) The phrase **if** $x = 0$ **then** x **else** y is a perfectly legitimate phrase of type $\mathbf{var}[\mathbf{int}]$ though it does not denote a storage cell. The language guarantees, however, that the variables created by the primitive $\mathbf{new}[\delta]$ are always storage cells (also called “good variables” [65]). Their behavior is characterized as follows:

Definition. A trace t in $|\dagger var[\delta]|$ is called a (*storage*) *cell trace* if

$$\begin{aligned} t = \langle \dots, get.i, get.i', \dots \rangle &\implies i = i' \\ t = \langle \dots, put.i, get.i', \dots \rangle &\implies i = i' \end{aligned}$$

For every data value $i \in |\delta|$, define an element

$$cell_i = \{ t \in |\dagger var[\delta]| : \langle put.i \rangle \cdot t \text{ is a cell trace} \}$$

This element is called the *cell trace set with initial value* i .

Note that each $cell_i$ is an active element of $\dagger var[\delta]$.

4.2. Functions on Objects

Having postulated a notion of objects for modelling the types of Algol, we examine what functions are appropriate for such objects. As an example, consider the phrase

$$x : \mathbf{var}[int] \vdash [\text{val} = x, \text{inc} = (x := x + 1)] : \mathbf{counter}$$

which constructs a counter object from a variable object. The meaning of the phrase must be a function from variable objects to counter objects:

$$mkcounter : \dagger var[int] \rightarrow \dagger counter$$

The purpose of such a function is to determine the behavior of a counter object in terms of the behavior of a given variable object.

We have observed that objects are only sequentially usable, and we have already defined object spaces with built-in structure for sequential reuse of objects. Such objects are now usable just *once* in an object function, i.e., every trace of the output object must “come from” a unique trace of the input object. Therefore, we postulate:

THEESIS 3 *Object functions must be linear functions.*

Example: Consider the Algol function $inc : \mathbf{var}[int] \rightarrow \mathbf{comm}$ defined by

$$inc(x) = (x := x + 1)$$

It maps objects of type $\mathbf{var}[int]$ to objects of type \mathbf{comm} , i.e., it is a linear function of type $\dagger var[int] \rightarrow_L \dagger comm$. The linear map of this function consists of input-output pairs of the form:

$$\begin{array}{ll} \langle \rangle & \mapsto \langle \rangle \\ \langle \text{get}.i_1, \text{put}.(i_1 + 1) \rangle & \mapsto \langle * \rangle \\ \langle \text{get}.i_1, \text{put}.(i_1 + 1), \text{get}.i_2, \text{put}.(i_2 + 1) \rangle & \mapsto \langle * \rangle^2 \\ & \vdots \\ \langle \text{get}.i_1, \text{put}.(i_1 + 1), \dots, \text{get}.i_n, \text{put}.(i_n + 1) \rangle & \mapsto \langle * \rangle^n \\ & \vdots \end{array}$$

where i_1, \dots, i_n stand for arbitrary integers in $|int|$. The notation $\langle * \rangle^n$ means a sequence of n $*$'s.

This map must be understood as saying that whenever the command $inc(x)$ is demanded n times, an n -fold repetition of a get-put sequence is demanded on x , thereby causing the corresponding changes in the internal state of x .

Note that there is no requirement in the above map that each i_{k+1} is equal to $i_k + 1$. While this property is satisfied by storage cells, the meaning of inc should be applicable to *all* objects of type $\dagger var[int]$, not only storage cells. □

There is another ingredient to object functions. They produce an object behavior from the behaviors of their input objects, but other than this, they do not depend on anything else. They do not have any hidden state in which they can remember

information. So, if they simulate a particular operation of the output object by a series of operations on the input object, they should also simulate any future instance of the same operation by the same series of operations on the input object. Notice this in the linear map of the function *inc* in the above example. Every “*” in the output behavior “comes from” a sequence $\langle \text{get}.i, \text{put}.(i + 1) \rangle$ in the input behavior. This is independent of the position of “*” in the output trace. Functions that behave in this form may be described as being “passive”, “history-free”, or “regular”. This leads us to postulate:

THEESIS 4 *Object functions are regular functions.*

The notion of regular functions is formalized as follows:

Definition. Given object spaces $\dagger A$ and $\dagger B$, a *regular function* $f : \dagger A \rightarrow_R \dagger B$ is a linear function (viewed as a linear map) that satisfies:

1. $(s_1 \mapsto t_1), \dots, (s_n \mapsto t_n) \in f$ implies $(s_1 \cdots s_n \mapsto t_1 \cdots t_n) \in f$, and
2. $(s \mapsto t_1 \cdots t_n) \in f$ implies there exists a decomposition $s = s_1 \cdots s_n$ such that $(s_1 \mapsto t_1), \dots, (s_n \mapsto t_n) \in f$.

(The decomposition $s_1 \cdots s_n$ in the second condition is forced to be unique.) The definition captures the idea that the translations carried out by a regular function are time-independent. An output trace t_i can come from s_i some time in the “future” if and only if it can come from s_i at the “present”. Note also that a regular function maps prefix-closed elements of $\dagger A$ to prefix-closed elements of $\dagger B$. (Suppose $s \mapsto t \in f$ and t_1 is a prefix of t , i.e., there is t_2 such that $t = t_1 t_2$. Then, by condition 2, there is a prefix s_1 of s such that $s_1 \mapsto t_1 \in f$. So, if $x \in \dagger A$ is a prefix-closed element, $f(x)$ is also prefix-closed.)

It may be verified that the function *inc* of the above example is a regular function. An example of a non-regular linear function is the function *accumulate* : $\dagger \text{int} \rightarrow_L \dagger \text{int}$ with the following form of input-output pairs:

$$\langle i_1, i_2, \dots, i_n \rangle \mapsto \langle i_1, i_1 + i_2, \dots, \sum_{k=1}^n i_k \rangle$$

This is not regular because, for example, $\langle i_1 \rangle \mapsto \langle i_1 \rangle$ and $\langle i_2 \rangle \mapsto \langle i_2 \rangle$ are in *accumulate*, but not $\langle i_1, i_2 \rangle \mapsto \langle i_1, i_2 \rangle$. This function remembers information from past uses; so it is not regular.

A regular function is a kind of homomorphism. See Appendix A for a formal treatment of this structure. Just as homomorphisms on free algebras are uniquely determined by their action on a set of generators, regular functions are uniquely determined by simpler linear functions.

PROPOSITION 6 *There is an order-isomorphism $\dagger A \rightarrow_L B \cong \dagger A \rightarrow_R \dagger B$.*

If $f : \dagger A \rightarrow_L B$ is a linear function, define the corresponding regular function $\hat{f} : \dagger A \rightarrow_R \dagger B$ by

$$\hat{f} = \{ s_1 \cdots s_n \mapsto \langle b_1, \dots, b_n \rangle : n \geq 0 \wedge s_1 \mapsto b_1, \dots, s_n \mapsto b_n \in f \} \quad (12)$$

Conversely, if $g : \dagger A \rightarrow_R \dagger B$ is a regular function, we obtain a linear function $\check{g} : \dagger A \rightarrow_L B$ by

$$\check{g} = \{ s \mapsto b : s \mapsto \langle b \rangle \in g \} \quad (13)$$

It is easy to verify that these two constructions are mutually inverse. They are evidently monotone (in the inclusion ordering of linear maps).

We call \hat{f} the *regular extension* of f and \check{g} the *linear pattern* of g . Note that the regular function *inc* of the earlier example has the linear pattern

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

The regular function itself is obtained by iteration of this pattern as defined by (12).

As described in Section 3, such a linear pattern determines a function at the level of objects. See definition (10). If $(Q, \alpha \subseteq Q \times |var[\delta]| \times Q)$ is a variable object, the function *inc* applied to this object yields a command object $(Q, \alpha' \subseteq Q \times |comm| \times Q)$ given by:

$$(q, *, q') \in \alpha' \iff \exists i. (q, \langle \text{get}.i, \text{put}.(i+1) \rangle, q') \in \bar{\alpha}$$

In particular, if the variable object is the storage cell (6) then the resulting command object has the transition map:

$$(i, *, i') \in \alpha' \iff i' = i + 1$$

Thus, a linear pattern represents, in a compact form, the effect of an object function on all possible argument objects.

Multiple arguments

The above considerations generalize to object functions with multiple arguments in a straightforward fashion. Consider a function with argument objects of types $\dagger A_1, \dots, \dagger A_n$ and result of type $\dagger B$. This should be visualized as in Fig. 1(b) except that there are multiple inner objects involved in building the result object. Carrying out an operation of the result object involves performing some number of operations on potentially all the inner objects. Therefore, the function must first be a linear function of type $\dagger A_1 \otimes \cdots \otimes \dagger A_n \rightarrow_L \dagger B$. The elements of its linear map are pairs of the form $\mathbf{s} \mapsto t$ where \mathbf{s} is a tuple of traces s_1, \dots, s_n . Secondly, the function must be a regular function in the following sense:

Definition. A *regular map* $f : \dagger A_1 \otimes \cdots \otimes \dagger A_n \rightarrow_R \dagger B$ is a linear map satisfying:

1. $s_1 \mapsto t_1, \dots, s_n \mapsto t_n \in f$ implies $(s_1 \cdots s_n \mapsto t_1 \cdots t_n) \in f$, and

Table 7. Examples of object functions

id_A	$: A \rightarrow A$	$= \{ \langle a \rangle \mapsto a : a \in A \}$
$g \circ f$	$: A \rightarrow C$	$= \{ s \mapsto c : \exists t. s \mapsto t \in \hat{f} \wedge t \mapsto c \in g \}$
$f(x)$	$: B$	$= \{ b \in B : \exists a_1, \dots, a_n \in x. \langle a_1, \dots, a_n \rangle \mapsto b \in f \}$
$+$	$: exp[int] \times exp[int] \rightarrow exp[int]$	$= \{ \langle \text{fst}.i, \text{snd}.j \rangle \mapsto i + j : i, j \in exp[int] \}$
$plus$	$: exp[int], exp[int] \rightarrow exp[int]$	$= \{ \langle \langle i \rangle, \langle j \rangle \rangle \mapsto i + j : i, j \in exp[int] \}$
$\text{fst}_{A,B}$	$: A \times B \rightarrow A$	$= \{ \langle \text{fst}.a \rangle \mapsto a : a \in A \}$
$\text{snd}_{A,B}$	$: A \times B \rightarrow B$	$= \{ \langle \text{snd}.b \rangle \mapsto b : b \in B \}$
$\langle f, g \rangle$	$: \mathbf{X} \rightarrow A \times B$	$= \{ s \mapsto \text{fst}.a : s \mapsto a \in f \} \cup \{ s \mapsto \text{snd}.b : s \mapsto b \in g \}$
discard_A	$: A \rightarrow \mathbf{1}$	$= \{ \langle \rangle \mapsto * \}$
$\text{fix}_A[f]$	$: A$	$= \{ a \in A : \exists n \geq 0. \langle \rangle \mapsto a \in f^n \}$

2. $(s \mapsto t_1 \cdots t_n) \in f$ implies there exists a decomposition $s = s_1 \cdots s_n$ such that $(s_1 \mapsto t_1), \dots, (s_n \mapsto t_n) \in f$.

where concatenation $s_1 \cdots s_n$ on tuples of traces is defined pointwise.

Note that the product $\dagger A_1 \times \cdots \times \dagger A_n$ would not be useful here since a linear function can only use information from *one* of the components of a product. (Cf. the discussion of Table 4.)

Proposition 6 generalizes as:

$$\left(\bigotimes_i \dagger A_i \right) \rightarrow_L B \cong \left(\bigotimes_i \dagger A_i \right) \rightarrow_R \dagger B$$

So, we can still express regular functions by linear patterns.

Notation. From now on, we say that f is an *object function* from A_1, \dots, A_n to B and write $f : A_1, \dots, A_n \rightarrow B$ to mean that f is a linear function $\dagger A_1 \otimes \cdots \otimes \dagger A_n \rightarrow_L B$. (It uniquely determines a regular function $\hat{f} : \dagger A_1 \otimes \cdots \otimes \dagger A_n \rightarrow_R \dagger B$.) We use bold face letters \mathbf{A} to range over sequences of the form A_1, \dots, A_n . Sometimes we use identifiers as labels for the arguments. In that case, we write $f : (x_1 : A_1), \dots, (x_n : A_n) \rightarrow B$ to mean that f is a linear function from a labelled tensor product: $[x_1 : \dagger A_1 \otimes \cdots \otimes x_n : \dagger A_n] \rightarrow_L B$.

This terminology is convenient because A_i 's and B directly correspond to Algol types and we implicitly understand that we intend for functions to operate on *objects* of these types. The expert reader would note that we are using a multicategory structure on object spaces and object functions [33]. An alternative view in terms of coalgebras may be found in Appendix A.

Table 7 gives examples of object functions. Note that the composition of $f : A \rightarrow B$ and $g : B \rightarrow C$ is obtained by the composition $g \circ \hat{f}$ of linear functions. One can similarly define composition for object functions with multiple arguments. The application of an object function $f : A \rightarrow B$ to an element $x \in A$ is defined

Table 8. Interpretation of constants

<i>skip</i>	: <i>comm</i>	= { $*$ }
<i>seq</i>	: <i>comm</i> \times <i>comm</i> \rightarrow <i>comm</i>	= { $\langle \text{fst.}*, \text{snd.}*\rangle \mapsto *$ }
<i>par</i>	: <i>comm</i> , <i>comm</i> \rightarrow <i>comm</i>	= { $\langle (*), (*)\rangle \mapsto *$ }
<i>cond</i> _A	: <i>exp</i> [<i>bool</i>] \times <i>A</i> \times <i>A</i> \rightarrow <i>A</i>	= { $\langle \text{fst.tt}, \text{snd.a}\rangle \mapsto a : a \in A $ } \cup { $\langle \text{fst.ff}, \text{third.a}\rangle \mapsto a : a \in A $ }
<i>assign</i> _{δ}	: <i>var</i> [δ] \times <i>exp</i> [δ] \rightarrow <i>comm</i>	= { $\langle \text{snd.i}, \text{fst.put.i}\rangle \mapsto * : i \in \delta $ }
<i>deref</i> _{δ}	: <i>var</i> [δ] \rightarrow <i>exp</i> [δ]	= { $\langle \text{get.i}\rangle \mapsto i : i \in \delta $ }
<i>new</i> _{δ}	: (<i>var</i> [δ] \rightarrow <i>comm</i>) \rightarrow <i>comm</i>	= { $\langle s \mapsto *\rangle \mapsto * : s \in \text{cell}_{\text{init}}[\delta]$ }

using the same principle. Note also the distinction between the two kinds of object functions $A, B \rightarrow C$ and $A \times B \rightarrow C$. The former kind of function acts on two separate objects with operations of types A and B respectively. The latter acts on a single object with two operations of types A and B respectively. The element $\text{fix}_A[f]$ denotes the least fixed point of an object function $f : A \rightarrow A$, i.e., the least element $x \in A$ such that $f(x) = x$.

In Table 8, we list the functions involved in interpreting the constants of interference-controlled Algol. The function *seq* (which interprets “;”) acts on an object with two command operations and produces a command that runs the two commands in sequence. The function *par* (which interprets “||”) acts on two independent command objects and runs them in parallel (by extracting a token $*$ from each). The remaining functions can be understood similarly, except for *new* _{δ} which involves a higher-order type discussed below.

Function types

An object function of type $A \rightarrow B$ is a linear function $\dagger A \rightarrow_L B$. Since all such functions are representable by the coherent space $\dagger A \multimap B$, this directly gives us a representation of the function space for interference-controlled Algol. Explicitly, define the function space $A \Rightarrow B$ as a coherent space with:

$$\begin{aligned}
|A \Rightarrow B| &= \{ s \mapsto b : s \in |\dagger A| \wedge b \in |B| \} \\
(s \mapsto b) \circ_{A \Rightarrow B} (s' \mapsto b') &\iff (s \circ_{\dagger A} s' \iff b \circ_B b' \wedge \\
&\quad s \circ_{\dagger A} s' \wedge b = b' \iff s = s')
\end{aligned}$$

The function space satisfies an order-isomorphism:

$$\mathbf{X}, A \rightarrow B \cong \mathbf{X} \rightarrow (A \Rightarrow B) \tag{14}$$

as a direct consequence of isomorphism (11). Suppose \mathbf{X} is the sequence X_1, \dots, X_n . Then, an object function $\mathbf{X}, A \rightarrow B$ is a linear function $\dagger X_1 \otimes \dots \otimes \dagger X_n \otimes \dagger A \rightarrow_L B$. By (11), such linear functions are one-to-one with linear functions $\dagger X_1 \otimes \dots \otimes \dagger X_n \rightarrow_L (\dagger A \multimap B)$ and these are nothing but object functions $\mathbf{X} \rightarrow (A \Rightarrow B)$.

The curry combinator that maps $f : \mathbf{X}, A \rightarrow B$ to $(\text{curry } f) : \mathbf{X} \rightarrow (A \Rightarrow B)$ is given by

$$\text{curry } f = \{ \mathbf{r} \mapsto (s \mapsto b) : (\mathbf{r}, s \mapsto b) \in f \}$$

By setting \mathbf{X} to the empty sequence, we obtain $A \rightarrow B \cong \rightarrow (A \Rightarrow B)$. Thus, every object function $A \rightarrow B$ can be regarded as element of $A \Rightarrow B$ which, in turn, corresponds to a regular element of $\dagger(A \Rightarrow B)$. But, there are other (non-regular) elements of $\dagger(A \Rightarrow B)$ which model dynamic objects with function-type methods.

4.3. Semantics

The foregoing discussion gives us enough tools to define the semantics of the basic phrases of interference-controlled Algol. This corresponds to the language defined by the first three lines of Table 1. While we discuss passivity in the next section, which is essential for interpreting the structural rules, we show the semantics of the basic phrases here to make the discussion concrete.

The types of interference-controlled Algol are interpreted as coherent spaces:

$$\begin{aligned} \llbracket \mathbf{var}[\delta] \rrbracket &= \text{var}[\delta] & \llbracket \theta_1 \times \theta_2 \rrbracket &= \llbracket \theta_1 \rrbracket \times \llbracket \theta_2 \rrbracket \\ \llbracket \mathbf{exp}[\delta] \rrbracket &= \text{exp}[\delta] & \llbracket \theta_1 \rightarrow \theta_2 \rrbracket &= \llbracket \theta_1 \rrbracket \Rightarrow \llbracket \theta_2 \rrbracket \\ \llbracket \mathbf{comm} \rrbracket &= \text{comm} \end{aligned}$$

The semantics of phrases is given by induction on their type derivations. Note that a phrase P has a typing of the form

$$x_1 : \theta_1, \dots \mid \dots, x_n : \theta_n \vdash P : \theta$$

where x_1, \dots, x_n are assumed to be distinct identifiers and the order of the typing assumptions is immaterial (within their zones). The meaning of such a phrase, written

$$\llbracket x_1 : \theta_1, \dots \mid \dots, x_n : \theta_n \vdash P : \theta \rrbracket$$

is an object function of type

$$(x_1 : \llbracket \theta_1 \rrbracket), \dots, (x_n : \llbracket \theta_n \rrbracket) \rightarrow \llbracket \theta \rrbracket$$

This, in turn, means that it is a linear map of type

$$[x_1 : \dagger \llbracket \theta_1 \rrbracket \otimes \dots \otimes x_n : \dagger \llbracket \theta_n \rrbracket] \rightarrow_L \llbracket \theta \rrbracket$$

(Note that the zones of the free identifiers do not affect the form of the map. This will be refined using a treatment of passivity in Sec. 5.) The linear map is a coherent set of input-output pairs each of the form $[x_1 \rightarrow s_1, \dots, x_n \rightarrow s_n] \mapsto a$. We call the input part of such a pair a “trace environment.” The metavariable η is used to range

Table 9. Semantic interpretation of phrase rules

id	$[[\Pi \mid \Gamma, x : \theta \vdash x : \theta]]$	$= \{ \eta_0[x \mapsto \langle a \rangle] \mapsto a \}$
$\times \mathcal{I}$	$[[\Pi \mid \Gamma \vdash (P, Q) : \theta_1 \times \theta_2]]$	$= \{ \eta \mapsto \text{fst}.a : \eta \mapsto a \in [[\Pi \mid \Gamma \vdash P : \theta_1]] \} \cup$ $\{ \eta \mapsto \text{snd}.a : \eta \mapsto a \in [[\Pi \mid \Gamma \vdash Q : \theta_2]] \}$
$\times 1 \mathcal{E}$	$[[\Pi \mid \Gamma \vdash \text{fst}(P) : \theta_1]]$	$= \{ \eta \mapsto a : \eta \mapsto \text{fst}.a \in [[\Pi \mid \Gamma \vdash P : \theta_1 \times \theta_2]] \}$
$\times 2 \mathcal{E}$	$[[\Pi \mid \Gamma \vdash \text{snd}(P) : \theta_2]]$	$= \{ \eta \mapsto a : \eta \mapsto \text{snd}.a \in [[\Pi \mid \Gamma \vdash P : \theta_1 \times \theta_2]] \}$
$\rightarrow \mathcal{I}$	$[[\Pi \mid \Gamma \vdash \lambda x : \theta. P : \theta \rightarrow \theta']]$	$= \{ \eta \mapsto (s \mapsto b) : \eta \oplus [x \mapsto s] \mapsto b \in [[\Pi \mid \Gamma, x : \theta \vdash P : \theta']] \}$
$\rightarrow \mathcal{E}$	$[[\Pi, \Pi' \mid \Gamma, \Gamma' \vdash PQ : \theta']]$	$= \{ \eta \oplus \eta' \mapsto b : \exists s. \eta \mapsto (s \mapsto b) \in [[\Pi \mid \Gamma \vdash P : \theta \rightarrow \theta']] \wedge$ $\eta' \mapsto s \in [[\Pi' \mid \Gamma' \vdash Q : \theta]] \}$

over such trace environments. The symbol η_0 denotes a trace environment where all variables are mapped to $\langle \rangle$, $\eta[x \mapsto s]$ denotes η with the x component updated to s , and $\eta \oplus \eta'$ denotes the join of two trace environments η and η' with disjoint domains. If η and η' are trace environments with identical domains, then $\eta \cdot \eta'$ denotes the trace environment with x mapped to $\eta(x) \cdot \eta'(x)$.

Table 9 gives the semantic interpretations of the basic phrases. For the most part, the combinators used in the interpretation have already been mentioned earlier. Recall that the notation $[[\Pi' \mid \widehat{\Gamma'} \vdash Q : \theta]]$ means the regular extension of the object function.

The interpretation of the primitive phrases is obtained by applying the constants shown in Table 8. For example, the meaning of $[[\Pi \mid \Gamma \vdash C_1; C_2 : \mathbf{comm}]]$ is obtained by $seq \circ \langle [[\Pi \mid \Gamma \vdash C_1 : \mathbf{comm}]], [[\Pi \mid \Gamma \vdash C_2 : \mathbf{comm}]] \rangle$. For convenience, we show some of these derived meanings in Table 10. Note in particular the difference between sequential and parallel composition. The interpretation of $\mathbf{new}[\delta] x.C$ is to apply the meaning of the function term $\lambda x.C$ to the object behavior $cell_{init[\delta]}$ (which is the behavior of a storage cell with initial value $init[\delta]$). The denotation of $\lambda x.C$ then uses a unique trace s from this behavior to run the command C .

Example: Consider the Algol function

$$\begin{aligned} mkcounter &: \mathbf{var}[int] \rightarrow \mathbf{counter} \\ mkcounter(x) &= [\mathbf{val} = \mathbf{deref } x, \mathbf{inc} = (x := x + 1)] \end{aligned}$$

Its meaning is an object function of type $var[int] \rightarrow counter$ containing the pairs

$$\begin{aligned} \langle \mathbf{get}.i \rangle &\mapsto \mathbf{val}.i \\ \langle \mathbf{get}.i, \mathbf{put}.(i + 1) \rangle &\mapsto \mathbf{inc}.* \end{aligned}$$

for all $i \in |int|$. The corresponding regular function is obtained by iterating the above pattern. For example, it has input-output pairs of the form:

$$\langle \mathbf{get}.i_1, \mathbf{get}.i_2, \mathbf{put}.(i_2 + 1), \mathbf{get}.i_3 \rangle \mapsto \langle \mathbf{val}.i_1, \mathbf{inc}.*, \mathbf{val}.i_3 \rangle$$

Table 10. Interpretation of primitive phrases

$\llbracket \Pi \mid \Gamma \vdash E_1 + E_2 : \mathbf{exp[int]} \rrbracket$	$= \{ \eta \cdot \eta' \mapsto i + j : \eta \mapsto i \in \llbracket \Pi \mid \Gamma \vdash E_1 : \mathbf{exp[int]} \rrbracket \wedge \eta' \mapsto j \in \llbracket \Pi \mid \Gamma \vdash E_2 : \mathbf{exp[int]} \rrbracket \}$
$\llbracket \Pi \mid \Gamma \vdash \mathbf{skip} : \mathbf{comm} \rrbracket$	$= \{ \eta_0 \mapsto * \}$
$\llbracket \Pi \mid \Gamma \vdash C_1 ; C_2 : \mathbf{comm} \rrbracket$	$= \{ \eta \cdot \eta' \mapsto * : \eta \mapsto * \in \llbracket \Pi \mid \Gamma \vdash C_1 : \mathbf{comm} \rrbracket \wedge \eta' \mapsto * \in \llbracket \Pi \mid \Gamma \vdash C_2 : \mathbf{comm} \rrbracket \}$
$\llbracket \Pi, \Pi' \mid \Gamma, \Gamma' \vdash C_1 \parallel C_2 : \mathbf{comm} \rrbracket$	$= \{ \eta \oplus \eta' \mapsto * : \eta \mapsto * \in \llbracket \Pi \mid \Gamma \vdash C_1 : \mathbf{comm} \rrbracket \wedge \eta' \mapsto * \in \llbracket \Pi' \mid \Gamma' \vdash C_2 : \mathbf{comm} \rrbracket \}$
$\llbracket \Pi \mid \Gamma \vdash \mathbf{new}[\delta] x. C : \mathbf{comm} \rrbracket$	$= \{ \eta \mapsto * : \exists s \in \mathit{cell}_{\mathit{init}}[\delta]. \eta \oplus [x \mapsto s] \mapsto * \in \llbracket \Pi \mid \Gamma, x : \mathbf{var}[\delta] \vdash C : \mathbf{comm} \rrbracket \}$
$\llbracket \Pi \mid \Gamma \vdash V := E : \mathbf{comm} \rrbracket$	$= \{ \eta' \cdot \eta \mapsto * : \exists i. \eta \mapsto \mathbf{put}.i \in \llbracket \Pi \mid \Gamma \vdash V : \mathbf{var}[\delta] \rrbracket \wedge \eta' \mapsto i \in \llbracket \Pi \mid \Gamma \vdash E : \mathbf{exp}[\delta] \rrbracket \}$
$\llbracket \Pi \mid \Gamma \vdash \mathbf{deref} V : \mathbf{exp}[\delta] \rrbracket$	$= \{ \eta \mapsto i : \eta \mapsto \mathbf{get}.i \in \llbracket \Pi \mid \Gamma \vdash V : \mathbf{var}[\delta] \rrbracket \}$

If we apply this function to the cell trace set shown in Fig. 2(e), we obtain the counter trace set shown in Fig. 2(b). \square

Example: As an example of higher-type objects, consider the following object constructor for bank accounts:

$$\begin{aligned} \mathbf{account} &= [\mathbf{bal} : \mathbf{exp[int]} \times \mathbf{dep} : \mathbf{exp[int]} \rightarrow \mathbf{comm} \times \mathbf{wd} : \mathbf{exp[int]} \rightarrow \mathbf{comm}] \\ \mathit{mkaccount} &: \mathbf{var[int]} \rightarrow \mathbf{account} \\ \mathit{mkaccount}(x) &= [\mathbf{bal} = \mathbf{deref} x, \mathbf{dep} = (\lambda a. x := x + a), \mathbf{wd} = (\lambda a. x := x - a)] \end{aligned}$$

(We allow the balance to go negative, for simplicity.) The operations dep and wd are of function types. We have a corresponding object space

$$\mathit{account} = [\mathbf{bal} : \mathit{exp[int]} \times \mathbf{dep} : \mathit{exp[int]} \Rightarrow \mathit{comm} \times \mathbf{wd} : \mathit{exp[int]} \Rightarrow \mathit{comm}]$$

The meaning of $\mathit{mkaccount}$ is an object function:

$$\begin{aligned} \mathit{mkaccount} &: \mathit{var[int]} \rightarrow \mathit{account} \\ \mathit{mkaccount} &= \{ \langle \mathbf{get}.n \rangle \mapsto \mathbf{bal}.n : n \in |\mathit{int}| \} \cup \\ &\quad \{ \langle \mathbf{get}.n, \mathbf{put}.(n+i) \rangle \mapsto \mathbf{dep}.(i \mapsto *) : n, i \in |\mathit{int}| \} \cup \\ &\quad \{ \langle \mathbf{get}.n, \mathbf{put}.(n-i) \rangle \mapsto \mathbf{wd}.(i \mapsto *) : n, i \in |\mathit{int}| \} \end{aligned}$$

Again, we obtain an account object by applying the regular extension of $\mathit{mkaccount}$ to an integer cell with initial value 0. \square

5. Passivity

The framework defined in Section 4 models interference-controlled Algol without any notion of passive values. This has some undesirable consequences:

1. All operations of an object must be completely sequenced, including operations that only read information from an object.

Table 11. Passive tokens for active-passive spaces

$ exp[\delta] _\wp$	$=$	$ exp[\delta] $
$ comm _\wp$	$=$	\emptyset
$ var[\delta] _\wp$	$=$	$\{get.i : i \in \delta \}$
$ A \times B _\wp$	$=$	$\{fst.a : a \in A _\wp\} \cup \{snd.b : b \in B _\wp\}$
$ A \otimes B _\wp$	$=$	$\{(a, b) : a \in A _\wp \wedge b \in B _\wp\}$
$ \top _\wp$	$=$	$ \top = \emptyset$
$ \mathbf{1} _\wp$	$=$	$ \mathbf{1} = \{*\}$

2. Object spaces contain some traces that are not realizable, such as $\langle get.0, get.1 \rangle$ in $|\dagger var[int]|$ or $\langle bal.100, bal.200 \rangle$ in $|\dagger account|$.

To solve the first problem, Reynolds introduced a notion of passive types. We will see that, by modelling passive types, we can solve the second (semantic) problem as well.

5.1. Active-passive spaces

Our main use of coherent spaces is for modelling the transitions of state machines that make up objects. We expect that some of these transitions are “passive” in that they only read information from the object without altering its state. This suggests that we must delineate, in each coherent space, certain tokens as “passive tokens.”

Definition. An *active-passive (coherent) space* A is a coherent space together with a designated set of tokens $|A|_\wp \subseteq |A|$. The members of $|A|_\wp$ are called *passive tokens* and the others *active tokens*. If all tokens of the active-passive space are passive, i.e., $|A|_\wp = |A|$, then it is called a *passive space*.

Table 11 lists the passive tokens of coherent spaces that we need for modelling interference-controlled Algol. All these spaces will now be regarded as active-passive coherent spaces. It is worth noting that $exp[\delta]$ is a passive space, which corresponds to the fact that expressions of Idealized Algol do not have side effects. The space **comm**, on the other hand, has no passive tokens. Its unique token $*$ is meant to change state. Note also that the *get* components of $var[\delta]$ are passive. The unit domains \top and $\mathbf{1}$ are passive and the constructions \times and \otimes preserve passivity, i.e., $A \times B$ and $A \otimes B$ are passive whenever A and B are passive. The constructions $\dagger A$ and $A \Rightarrow B$ are different from the corresponding constructions for coherent spaces. They are discussed in detail below.

Note that, for any active-passive space A , there exists a (*universal*) *passive subspace* obtained by selecting just the passive tokens of A . We denote this subspace by $\wp A$. Formally, $|\wp A| = |\wp A|_\wp = |A|_\wp$ and the consistency relation $\circ_{\wp A}$ is the

corresponding restriction of \circlearrowleft_A . It is obviously a passive space. Taking the passive subspace of an already passive space has no effect, i.e., $\wp P = P$ for any passive space P . In particular, $\wp\wp A = \wp A$.

An object $(Q, \alpha \subseteq Q \times |A| \times Q)$ for an active-passive space A has an additional condition on its transition map:

$$(q, a, q') \in \alpha \wedge a \in |A|_{\wp} \implies q = q'$$

This represents the intuition that passive transitions do not change state. Most of our development from Sections 3 and 4 can be carried over to active-passive spaces. There are only two changes. We modify the definition of object spaces using the fact that passive transitions do not change the state. And, we restrict object functions to account for passivity.

Active-passive object spaces

To arrive at the new definition of object spaces, we make the following observations:

1. If successive operations on an object are passive operations, their relative *order* is insignificant, i.e., we want to regard the sequences $\langle a, a' \rangle$ and $\langle a', a \rangle$ as the *same* trace whenever a and a' are passive tokens.
2. If successive operations on an object are identical passive operations, their *number* is insignificant, i.e., we want to regard the sequences $\langle a, a \rangle$ and $\langle a \rangle$ as the same trace whenever a is a passive token.

These identifications have a deeper significance than just ignoring order and number. Since the tokens represent entire operations, not merely atomic events, the identifications have the effect that passive operations on objects can in fact be done concurrently. Thus, to model passivity, we revise Thesis 1 to the effect that objects can in general be used sequentially, except that *multiple passive operations can be done concurrently*. This matches closely with the syntax of interference-controlled Algol, in particular the *Contr* type rule.

The identifications mentioned above are formalized as follows:

Definition. Let A be an active-passive space. The *trace monoid* of A is the quotient monoid $|A|^*/\equiv$ where \equiv is the least monoid congruence generated by the equivalence relation:

$$\begin{aligned} \langle a, a' \rangle &\equiv \langle a', a \rangle && \text{for all } a, a' \in |A|_{\wp} \\ \langle a, a \rangle &\equiv \langle a \rangle && \text{for all } a \in |A|_{\wp} \end{aligned}$$

An element of $|A|^*/\equiv$ is an equivalence class of sequences, called an (*active-passive*) *trace*. The equivalence class containing a sequence s is denoted $[s]$. The *unit trace* (or *empty trace*) is $[\langle \rangle]$. The *multiplication* of traces is defined by $[s] \cdot [s'] = [ss']$. (We often omit writing “.”.)

Traces of this kind (and more general ones) have been studied extensively in the context of Petri nets [1], [36]. One useful result from this theory is the existence of so-called Foata-normal form. Applied to the current situation, it means that every trace $[s]$ can be written in the form $[s_1] \cdot [s_2] \cdots [s_n]$, where each $[s_i]$ is nonempty and the consecutive segments alternate between active and passive traces. Note that an active trace is a singleton equivalence class (tokens cannot be permuted). On the other hand, a passive trace is an equivalence class of all permutations. So, an active trace is essentially a sequence and a passive trace is essentially a finite set. The Foata-normal form of an active-passive trace can thus be regarded as a formal product of the form $p_0 a_1 p_1 \cdots a_n p_n$ where the a_i 's are active tokens and p_i 's are (possibly empty) finite sets of passive tokens.

In examples, we simply write traces as sequences with the implicit understanding that consecutive passive tokens can be freely permuted.

Definition. Given an active-passive coherent space A , the active-passive coherent space $\dagger A$ is defined by

- the token set $|\dagger A|$ being the set of all *coherent* traces over A , i.e., traces $p_0 a_1 p_1 \cdots a_n p_n$ in Foata-normal form where each p_i is a finite coherent set,
- passive atoms $|\dagger A|_{\emptyset}$ being coherent traces with no active tokens (i.e., $n = 0$), and
- consistency relation given by $p_0 a_1 p_1 \cdots a_n p_n \circ_{\dagger A} p'_0 a'_1 p'_1 \cdots a'_m p'_m$ iff $p_0 \circ p'_0$ and, for all $i = 1, \dots, \min(n, m)$,

$$\langle a_1, \dots, a_{i-1} \rangle = \langle a'_1, \dots, a'_{i-1} \rangle \implies a_i \circ_A a'_i \wedge p_i \circ p'_i$$

where $p \circ p'$ means $\forall c \in p, c' \in p'. c \circ_A c'$.

The consistency relation represents the intuition that only active transitions from the past can have a causal effect on future transitions. A coherent trace is a trace that is consistent with itself. For example, $\langle \text{get.0}, \text{get.1} \rangle$ is not a coherent trace of $\dagger \text{var}[\delta]$ because the tokens get.0 and get.1 are inconsistent even though their active past is identical (empty in both cases). Thus, our object spaces now contain only meaningful traces.

Note that the \dagger construction preserves passivity. For any passive space P , $\dagger P$ is passive. The traces of $\dagger P$ are finite coherent sets of tokens. So, for passive spaces, the \dagger construction coincides with Girard's $!$ construction [17], [19].¹² An important result of Girard's is that linear functions $!A \rightarrow_L B$ are order-isomorphic to stable functions $A \rightarrow_S B$. This result obviously carries over to passive object spaces as well. Linear functions $\dagger P \rightarrow_L A$ are order-isomorphic to stable functions $P \rightarrow_S A$ whenever P is a passive space.

Object Functions

If $f : A \rightarrow B$ models a function phrase and $f(x)$ is a passive value, then no active token of x could have been “used” in producing $f(x)$. A state change operation cannot be involved in a “pure reader” of state information. This leads to our final thesis regarding a model of interference-controlled Algol:

THESES 5 *The linear maps of object functions must be passivity-reflecting.*

We formalize this notion as follows:

Definition. A (*passivity-reflecting*) *linear function* $f : A \rightarrow_L B$ between active-passive spaces is a linear function of the underlying coherent spaces such that, for all $a \mapsto b \in \mu f$, $b \in |B|_\wp$ implies $a \in |A|_\wp$.

Regular maps $A \rightarrow_R B$ and object functions $\mathbf{A} \rightarrow B$ for active-passive spaces are now defined in the same way as for coherent spaces, using passivity-reflecting linear functions. The function space is defined to have passivity-reflecting tokens:

$$\begin{aligned} |A \Rightarrow B| &= \{ s \mapsto b : s \in |\dagger A| \wedge b \in |B| \wedge (b \in |B|_\wp \implies s \in |\dagger A|_\wp) \} \\ |A \Rightarrow B|_\wp &= \{ s \mapsto b \in |A \Rightarrow B| : b \in |B|_\wp \} \\ \bigcirc_{A \Rightarrow B} &\quad \text{same as for coherent spaces} \end{aligned}$$

The passive tokens of $A \Rightarrow B$ have the b component passive (and, hence, the s component passive as well).

To get the intuition behind these notions, let us relate them back to objects as state machines. If $f : A \rightarrow B$ is an object function (as a linear map), the corresponding function on objects maps objects with transition maps $\alpha_1 \subseteq Q \times |A| \times Q$ to objects with transition maps $\alpha_2 \subseteq Q \times |B| \times Q$ given by

$$(q, b, q') \in \alpha_2 \iff \exists s \in |\dagger A|. s \mapsto b \in f \wedge (q, s, q') \in \bar{\alpha}_1$$

If b is passive then $(q, b, q') \in \alpha_2$ implies $q = q'$. Hence, for the unique s such that $s \mapsto b \in f$, $(q, s, q') \in \bar{\alpha}_1$ implies $q = q'$. Since this must hold uniformly for all objects, we require that s should be passive.

Passive function space

To model Reynolds’s passive function type, we introduce another function space $A \Rightarrow_P B$.

Definition. For active-passive spaces A and B , the *passive function space* $A \Rightarrow_P B$ has the same tokens and consistency relation as $A \Rightarrow B$, but all its tokens are designated as passive tokens, i.e., $|A \Rightarrow_P B|_\wp = |A \Rightarrow_P B|$.

The passive function space has an isomorphism

$$\mathbf{P}, A \rightarrow B \cong \mathbf{P} \rightarrow (A \Rightarrow_P B) \quad (\text{for sequences of passive spaces } \mathbf{P})$$

The combinator *curry* is the same as for $A \Rightarrow B$. Note that

$$\text{curry } f = \{ \mathbf{p} \mapsto s \mapsto b : (\mathbf{p}, s \mapsto b) \in f \}$$

satisfies the passivity-preservation property only if \mathbf{p} is passive (since $s \mapsto b$ is a passive token of $A \Rightarrow_P B$). This is ensured by insisting that \mathbf{P} be passive in the above isomorphism.

By setting \mathbf{P} to be the empty sequence in the above isomorphism, we obtain $A \rightarrow B \cong \rightarrow (A \Rightarrow_P B)$. So, the elements of $A \Rightarrow_P B$ precisely correspond to object functions $A \rightarrow B$. There exists an object function $\text{rec}_A : (A \Rightarrow_P A) \rightarrow A$ such that $\text{rec}_A(f) = \text{fix}_A[f]$. This is used to interpret the recursion combinator rec_θ .

There are further theoretical properties one must note about passivity. These may be found in Appendix A.

5.2. Semantics

We now complete the definition of semantics for interference-controlled Algol using active-passive spaces. The semantics of phrases given in Sec. 4.3 using coherent spaces can be easily refined to active-passive coherent spaces. The interpretation of a type $\llbracket \theta \rrbracket$ is now understood to mean an active-passive space (using Table 11). The passive function type is interpreted by $\llbracket \theta \rightarrow_p \theta' \rrbracket = \llbracket \theta \rrbracket \Rightarrow_P \llbracket \theta' \rrbracket$.

The interpretation of a phrase $\Pi \mid \Gamma \vdash P : \theta$ was given as an object function of type $\llbracket \Pi \rrbracket, \llbracket \Gamma \rrbracket \rightarrow \llbracket \theta \rrbracket$ over coherent spaces. We now understand this to be an object function over active-passive spaces, i.e., a passivity-preserving linear map of type $(\otimes \llbracket \Pi \rrbracket) \otimes (\otimes \llbracket \Gamma \rrbracket) \rightarrow_L \llbracket \theta \rrbracket$. It can be verified that the interpretation in Table 9 gives passivity-preserving maps. Moreover, any input-output pair $\eta_1 \oplus \eta_2 \mapsto a$ in the interpretation of P (where η_1 is a trace environment of $\llbracket \Pi \rrbracket$ and η_2 one for $\llbracket \Gamma \rrbracket$) will only have passive traces in η_1 . An elegant way of stating this is that the interpretation of P is an object function of type

$$\wp \llbracket \Pi \rrbracket, \llbracket \Gamma \rrbracket \rightarrow \llbracket \theta \rrbracket$$

Again, this can be verified for the interpretation in Table 9. For the rule *Id*, the base case, η_1 maps all identifiers to $\langle \rangle$ which is a passive trace. The other inductive cases preserve this property.

Table 12 gives the interpretation of the structural rules. The interpretation of rule *Contr* says that, if an identifier x occurs multiply in independent contexts of a phrase, the information used from x is the union of the information used in each occurrence. Note that all these occurrences are passive uses of the identifier. So, the information used is represented by passive traces (finite coherent sets).

The remaining rules cause *no change* to the semantics of the phrase. In a sense, this is to be expected because we would not want to assign multiple interpretations

Table 12. Semantic interpretation of structural rules

Contr	$\llbracket \Pi, x : \theta \mid \Gamma \vdash [x/x_1, x/x_2]P : \theta' \rrbracket$	$= \{ \eta \oplus [x \rightarrow p_1 \cup p_2] \mapsto a : \eta \oplus [x_1 \rightarrow p_1, x_2 \rightarrow p_2] \mapsto a \in \llbracket \Pi, x_1 : \theta, x_2 : \theta \mid \Gamma \vdash P : \theta' \rrbracket \}$
Activate	$\llbracket \Pi \mid \Gamma, x : \theta \vdash P : \theta' \rrbracket$	$= \llbracket \Pi, x : \theta \mid \Gamma \vdash P : \theta' \rrbracket$
Passify	$\llbracket \Pi, x : \theta \mid \Gamma \vdash P : \phi \rrbracket$	$= \llbracket \Pi \mid \Gamma, x : \theta \vdash P : \phi \rrbracket$
Promote	$\llbracket \Pi \mid \vdash P : \theta_1 \rightarrow_p \theta_2 \rrbracket$	$= \llbracket \Pi \mid \vdash P : \theta_1 \rightarrow \theta_2 \rrbracket$
Derelict	$\llbracket \Pi \mid \Gamma \vdash P : \theta_1 \rightarrow \theta_2 \rrbracket$	$= \llbracket \Pi \mid \Gamma \vdash P : \theta_1 \rightarrow_p \theta_2 \rrbracket$

to the same phrase. We must verify, however, that these interpretations are of the right types. For the rule *Activate*, the meaning of P on the right is an object function of type $\wp[\Pi], x : \wp[\theta], [\Gamma] \mapsto \llbracket \theta' \rrbracket$. Clearly, it is also of type $\wp[\Pi], x : \llbracket \theta \rrbracket, [\Gamma] \mapsto \llbracket \theta' \rrbracket$. (We merely forget that the binding of x is passive.) For the rule *Passify*, the meaning of P on the right is an object function of type $\wp[\Pi], [\Gamma], x : \llbracket \theta \rrbracket \mapsto \llbracket \phi \rrbracket$. Since $\llbracket \phi \rrbracket$ is a passive space (and object functions are passivity-preserving), all input-output pairs in the meaning of P have only passive tokens. So, the meaning of P is also of type $\wp[\Pi], [\Gamma], x : \wp[\theta] \mapsto \llbracket \phi \rrbracket$.

For the rule *Promote*, the meaning of P on the right is an object function of type $\wp[\Pi] \rightarrow (\llbracket \theta_1 \rrbracket \Rightarrow \llbracket \theta_2 \rrbracket)$. Since the token set of $\llbracket \theta_1 \rrbracket \Rightarrow_p \llbracket \theta_2 \rrbracket$ is the same as that of $\llbracket \theta_1 \rrbracket \Rightarrow \llbracket \theta_2 \rrbracket$, the meaning is also of type $\wp[\Pi] \rightarrow (\llbracket \theta_1 \rrbracket \Rightarrow_p \llbracket \theta_2 \rrbracket)$. The verification of *Derelict* is similar.

Thus, we have shown:

PROPOSITION 7 *The meaning assigned to any type derivation of $\Pi \mid \Gamma \vdash P : \theta$ is an object function of type $\wp[\Pi], [\Gamma] \rightarrow \llbracket \theta \rrbracket$.*

Somewhat more technical is the following property of “coherence:”

PROPOSITION 8 *The meaning assigned to every type derivation of $\Pi \mid \Gamma \vdash P : \theta$ is the same object function.*

This can be proved using the same techniques as O’Hearn *et al.* [46]. Note that the structural rules other than *Contr* do not affect the meaning. So, their relative position in derivations does not affect the meaning either.

Operational adequacy

To show that the semantics defined here agrees with the operational behavior of interference-controlled Algol, we prove that it is *sound* and *adequate* with respect to the operational semantics:

PROPOSITION 9 *For any closed command phrase C , the denotation $\llbracket \vdash C : \mathbf{comm} \rrbracket$ is nonempty (has an input-output pair $[\] \mapsto *$) if and only if C terminates.*

Recall from Sec. 2 that the operational semantics of Algol is defined to work in two stages: reduction and execution. The command C terminates iff there is a closed command phrase C' such that $C \rightarrow^* C'$ and $([], C') \Downarrow []$. Our proof of operational adequacy is similarly factored into two stages. See Appendix B for details.

6. The Structure of Object Spaces

Having carried out a long analysis of object spaces and object functions, let us examine what results have been obtained.

The structure of Algol presents some paradoxical issues. Consider the type **comm** as a prototypical example of object types. The elements of type **comm** should correspond to observably distinct closed phrases of type **comm**. What phrases are there? Since a closed phrase does not have any free variable identifiers, the only observable effect of such a phrase is termination. So, there are precisely two closed phrases up to observational equivalence: **undef** and **skip**. A “good” semantics of Algol should thus have precisely two elements in $comm$.

Closed phrases of type **comm** \rightarrow **comm** should correspond to command phrases with a single free identifier $c : \mathbf{comm}$. Again, what phrases are there? We still have the diverging command **undef**. But, we also have some state to play with: whatever state the command c might act on. We can affect this state by running c some number of times. This gives phrases of the form

$$c^n \equiv \overbrace{c; \dots; c}^{n \text{ times}}$$

The various functions $\lambda c. c^n$ are observationally inequivalent. We can observe the difference between them by putting them in program contexts $P_n[]$ of the form

```

new[int]  $x$ .
   $x := 0$ ;
   $[](x := x + 1)$ ;
  if  $x = n$  then skip else undef

```

So, the elements of the function space $comm \Rightarrow comm$ should have the structure of flat naturals, N_{\perp} .¹³

Now, here is the paradox. If $comm$ has only two elements, how can $comm \Rightarrow comm$ have an infinite number of elements?¹⁴

This observation suggests that the notion of “elements” in a semantics of Algol is a delicate issue. There must be different kinds of elements which come to the fore depending on the context. The “elements” that correspond to meanings of closed phrases are *regular elements* mentioned in Sec. 4.1. Note that the meanings of closed phrases are functions of the form $f : \mathbf{1} \rightarrow_L A$. Such functions correspond to elements $x = \{a : * \mapsto a \in f\}$ of A . Their regular extensions $\hat{f} : \mathbf{1} \rightarrow_R \dagger A$ similarly correspond to elements x^* of $\dagger A$. So, the regular elements of $\dagger A$ form a domain isomorphic to the domain A , and both the domains represent meanings of

closed phrases. Armed with this notion, let us analyze some of the basic types of our model.

FACT 1 *There are precisely two regular elements in $\dagger comm$.*

The two elements are $diverge = \emptyset^* = \{\langle \rangle\}$ and $skip = \{*\}^*$.

FACT 2 *The elements of $(comm \Rightarrow comm)$ form a domain isomorphic to N_{\perp} .*

A token of $comm \Rightarrow comm$ is of the form $\langle * \rangle^n \mapsto *$ where $n \geq 0$. Denote such a token by \bar{n} . Since $\langle * \rangle^n \subset \langle * \rangle^m$ in $\dagger comm$, $\bar{n} \subset \bar{m}$ if and only if $n = m$. In other words, the consistency relation of $comm \Rightarrow comm$ is the identity relation making it a flat domain. Since each \bar{n} corresponds to a natural number n , the structure is that of N_{\perp} .

Note that this structure is right for interference-controlled Algol. The element \emptyset of $comm \Rightarrow comm$ models $\lambda c. \mathbf{undef}$ and an element $\{\bar{n}\}$ models $\lambda c. c^n$.

This fact is unprecedented in the semantics of imperative programs. All other known models fail to capture $comm \Rightarrow comm$ accurately. The model that comes closest is the parametricity-based model of [50] (as well as the related model of Sieber [72], [73]). In addition to the Algol functions mentioned above, the parametricity-based model contains functions expressible using the so-called “snap back” operator. The command snap-back combinator is a constant $\mathbf{try} : \mathbf{comm} \rightarrow_p \mathbf{comm}$ with the semantics that $\mathbf{try} C$ runs the command C and then snaps the state back to the original state. (By adding such a combinator, we can express new functions of the form $\lambda c. \mathbf{try} c^{n+m}; c^n$ which diverge for cases where $\lambda c. c^n$ would terminate.) Note that the snap-back operator is counter to one’s intuition that state changes are irreversible. The reason for the presence of such operators in the parametricity model (as well as other models) is that commands are modelled by state-to-state functions. We avoid this pitfall because, as stated in Section 3, we do not treat states as entities. They are at best derived attributes of objects, and do not play a central role in our model.

Notice how our model answers the puzzle mentioned at the beginning of this section. The regular elements of $\dagger comm$ are the values that are directly expressible. The other elements of $\dagger comm$ such as $\{\langle * \rangle^n\}$ remain hidden. They come to the fore in contexts where their difference becomes distinguishable (such as the argument positions of procedures).

One might wonder if the other elements of objects spaces are expressible in some other form. Indeed, an important class of elements, viz., *active elements* mentioned in Sec. 4.1, correspond to object behaviors. Such elements arise as the values of phrases with active free identifiers.

FACT 3 *The active elements of $\dagger comm$ form a domain isomorphic to $VNat$, the domain of “vertical” natural numbers (ordered by numerical order).*

The element corresponding to a natural number k is the set of all prefixes of $\langle * \rangle^k$. Denote it by \bar{k} . Such elements arise as the meanings of phrases:

$x : \mathbf{var}[\mathbf{int}] \vdash \mathbf{if } x \leq k \mathbf{ then } x := x + 1 \mathbf{ else undef} : \mathbf{comm}$

Applying the meaning of this phrase to $cell_0$, we obtain the element \tilde{k} .

Let us examine a second order type:

FACT 4 *The domain $(\mathit{comm} \Rightarrow \mathit{comm}) \Rightarrow \mathit{comm}$ is isomorphic to a subdomain of $P(N^*)$ that contains all subsets of N^* without any mutual prefixes.*

A token of $(\mathit{comm} \Rightarrow \mathit{comm}) \Rightarrow \mathit{comm}$ is of the form $\langle \bar{n}_1, \dots, \bar{n}_k \rangle \mapsto *$ where \bar{n}_i are tokens of $\mathit{comm} \Rightarrow \mathit{comm}$ mentioned previously. Two such tokens are consistent if and only if the sequences on the left are inconsistent, i.e., not prefixes of each other. So, every set of such tokens that excludes mutual prefixes on the left is an element of $(\mathit{comm} \Rightarrow \mathit{comm}) \Rightarrow \mathit{comm}$. The diligent reader would be able to verify that all finite elements of this domain are expressible by closed phrases. For example, the element $\{\langle \bar{n} \rangle \cdot s_1 \mapsto *, \langle \bar{n} \rangle \cdot s_2 \mapsto *, \langle \bar{m} \rangle \cdot t \mapsto *\}$, with $n \neq m$, is expressible by

```

λp. new[int] x.
  x := 0;
  p(x := x + 1);
  if x = n then g_s(p)
  else if x = m then g_t(p)
  else undef

```

where g_s and g_t are terms expressing the elements $\{s_1 \mapsto *, s_2 \mapsto *\}$ and $\{t \mapsto *\}$ respectively.

Regarding the semantics of passive types, we have the following observation. Notice that the fragment of interference-controlled Algol involving only passive types is a functional programming language. In this fragment, the distinction between the active and passive free identifiers vanishes because the free identifiers can freely move between the passive and active zones of typing judgements. So, the language is essentially the same as PCF (with product types). We have:

FACT 5 *The object-based semantics of the passive fragment coincides with the stable semantics of PCF [10].*

The semantics of this fragment involves only passive spaces, for which the function space $\dagger P \multimap Q$ is identical to the stable function space $[P \multimap_S Q]$.

More important to our concerns is the interaction between the active and passive sublanguages. With regard to this, we have:

FACT 6 *The domain $(\mathit{comm} \Rightarrow \mathit{exp}[\delta])$ is isomorphic to δ .*

This fact holds because of the passivity-preservation property of functions. The only passive token of $\dagger \mathit{comm}$ is the empty trace. Since $\mathit{exp}[\delta]$ is a passive space, the tokens of $\mathit{comm} \Rightarrow \mathit{exp}[\delta]$ are of the form $\langle \rangle \mapsto i$ for some $i \in |\delta|$. In other words, the elements of $\mathit{comm} \Rightarrow \mathit{exp}[\delta]$ are all constant functions. Note that this is indeed

the case for Idealized Algol. Any nontrivial use of a command in an expression would amount to a side effect and expressions of Idealized Algol do not have side effects.

Most other models of Algol fail to model $comm \Rightarrow exp[\delta]$ accurately. In fact, they support an expression snap-back combinator (more general than the **try** C operator mentioned above) of the form **do** C **result** E , where C is a command and E an expression. See the discussion in [50]. This combinator embeds commands inside expressions and causes the breakdown of most reasoning principles of expressions.¹⁵ The only other known model that satisfies Fact 6 is Tennent's model of specification logic [78]. Our techniques, however, seem markedly different from his.

We make a few remarks regarding the domain order of function spaces. Most semantic models of programming languages use continuous function spaces with pointwise order. In contrast, we are using linear functions (a subclass of stable functions) ordered by Berry's stable order. A question arises as to whether this is an appropriate choice. Indeed we find that the stable order matches quite closely with the semantic structure of interference-controlled Algol. Note that, in view of the discussion at the beginning of this section, the fully abstract model is not extensional. So, it cannot be order-extensional either. Moreover, the pointwise order is clearly inappropriate for the obvious possibilities. Given that $comm$ is a two point lattice, the continuous function space $[comm \rightarrow comm]$ would order the function $\lambda c.c$ below $\lambda c.skip$. But, they are distinguishable in the language. Even the function space $[VNat_{\perp} \rightarrow comm]$ does not have the right order to represent $\mathbf{comm} \rightarrow \mathbf{comm}$. In contrast, the stable order seems to be working correctly even up to second order functions, as evidenced by Facts 1-4.

On the other hand, when we look to passive types, the stable order results in well-known problems [30]. We can use this fact to our advantage. We understand that stable order is modelling history-sensitive computations. To combine it with the pointwise order required for history-free computations, we marry the two using the framework of bidomains. We indicate this briefly for the representation in terms of *bistructures* recently obtained by Plotkin and Winskel [55]. Recall that a bistructure is a 4-tuple $(|A|, \leq_A^L, \leq_A^R, \bigcirc_A)$ satisfying certain axioms. To this, we add another component $|A|_{\wp} \subseteq |A|$ for modelling passivity and require that $|A|_{\wp}$ be down-closed with respect to \leq_A^R and \geq_A^L . The (active-passive) coherent spaces for primitive types embed into (active-passive) bistructures in a straightforward fashion. We only need to define the construction $\dagger A$ for object spaces. This is as follows:

- tokens $|\dagger A|$ are active-passive traces $p_0 a_1 p_1 \cdots a_n p_n$ where each p_i is a finite (stable) configuration over $|A|_{\wp}$ and each a_i is an active token,
- $|\dagger A|_{\wp}$ and $\bigcirc_{\dagger A}$ are as for coherent spaces, and
- the orders \leq^l (for $l = L, R$) are given by $p_0 a_1 p_1 \cdots a_n p_n \leq^l q_0 b_0 q_1 \cdots b_m q_m$ iff $n = m$, each $a_i = b_i$ and each $p_i \leq^l q_i$ (the latter defined as for $!A$).

Note that the orders \leq^L and \leq^R reduce to identity for purely active bistructures like $\dagger comm$ and they reduce to the orders of $!A$ for purely passive bistructures like $\dagger exp[\delta]$. By using passivity-reflecting linear maps $\dagger A \rightarrow_L B$ as object functions, we obtain a model that properly restricts the coherent space model by incorporating pointwise order for passive computations.

7. Example Equivalences

An important application of a semantic model is to test the equivalence of program fragments. The accuracy of a model can be gauged by the variety of equivalences it can validate. Since the object-based model is accurate for many first-order types, we expect that most equivalences involving free identifiers of such types can be validated using it. In this section, we show several examples of such equivalences. In all of these, an unknown non-local procedure p is passed an object built from local variables.

Consider a variant of equivalence (1) from Sec. 1:

$$\begin{array}{l} \mathbf{new}[\mathbf{int}] x. \\ x := 0; \\ p(x := x + 1) \end{array} \quad \equiv \quad \begin{array}{l} \mathbf{new}[\mathbf{int}] x. \\ x := 0; \\ p(x := x - 1) \end{array}$$

where the free identifier p is of type $\mathbf{comm} \rightarrow \mathbf{comm}$. It is easy to see that the equivalence holds in the object-based model. Both sides of the equivalence denote the following object function of the type $(comm \Rightarrow comm) \rightarrow comm$:

$$\{ \langle \bar{n} \rangle \mapsto * : n \geq 0 \}$$

Recall from Sec. 6 that \bar{n} is short for $\langle * \rangle^n \mapsto *$, a token carrying the information that the procedure runs its argument n times.

A variant of the above example with a more practical interest is:

$$\begin{array}{l} \mathbf{new}[\mathbf{int}] x. \\ x := 0; \\ p[\mathbf{val} = x, \mathbf{inc} = (x := x + 1)] \end{array} \quad \equiv \quad \begin{array}{l} \mathbf{new}[\mathbf{int}] x. \\ x := 0; \\ p[\mathbf{val} = -x, \mathbf{inc} = (x := x - 1)] \end{array}$$

where $p : \mathbf{counter} \rightarrow \mathbf{comm}$. The procedure p is being passed two different implementations of a counter object. Since the two implementations have the same observable behavior, we expect the equivalence to hold.

The meanings of the two sides of the equivalence are:

$$\begin{array}{l} \{ [p \rightarrow \langle s \mapsto * \rangle] \mapsto * : s \in \widehat{mkcounter}_1(cell_0) \} \\ \{ [p \rightarrow \langle s \mapsto * \rangle] \mapsto * : s \in \widehat{mkcounter}_2(cell_0) \} \end{array}$$

where $mkcounter_1$ and $mkcounter_2$ are the meanings of the two counter phrases regarded as functions of a variable object, and $cell_0$ is the trace set of a storage cell with an initial value 0. So, the equivalence reduces to the equality of

$\widehat{mkcounter}_1(cell_0)$ and $\widehat{mkcounter}_2(cell_0)$. The two functions are given by the linear patterns:

$$\begin{aligned} \widehat{mkcounter}_1 &= \{ \langle \text{get}.i \rangle \mapsto \text{val}.i : i \in |int| \} \cup \\ &\quad \{ \langle \text{get}.i, \text{put}.(i+1) \rangle \mapsto \text{inc}.* : i \in |int| \} \\ \widehat{mkcounter}_2 &= \{ \langle \text{get}.i \rangle \mapsto \text{val}.(-i) : i \in |int| \} \cup \\ &\quad \{ \langle \text{get}.i, \text{put}.(i-1) \rangle \mapsto \text{inc}.* : i \in |int| \} \end{aligned}$$

It is easy enough to see that the two object functions produce the same counter trace set when applied to $cell_0$. But, let us see how one can show this formally.

THEOREM $\widehat{mkcounter}_1(cell_0) = \widehat{mkcounter}_2(cell_0)$.

Proof: The cell trace sets can be defined inductively as follows:

$$cell_i = \{ \langle \rangle \} \cup (\langle \text{get}.i \rangle \cdot cell_i) \cup (\bigcup_{j \in |int|} \langle \text{put}.j \rangle \cdot cell_j)$$

where \cdot denotes the obvious extension of multiplication to sets of traces. To be completely clear, we regard $cell$ as a function of type $|int| \rightarrow \dagger var[int]$. Since such functions form a cpo, the recursive definition has the standard interpretation as the least fixed point. A family of counter trace sets can be defined in a similar fashion:

$$cnt_i = \{ \langle \rangle \} \cup (\langle \text{val}.i \rangle \cdot cnt_i) \cup (\langle \text{inc}.* \rangle \cdot cnt_{i+1})$$

Then, by using fixpoint induction, we show:

$$\begin{aligned} \widehat{mkcounter}_1(cell_i) &= cnt_i \\ \widehat{mkcounter}_2(cell_{-i}) &= cnt_i \end{aligned}$$

for all $i \geq 0$. Evidently, the two functions are equal for $cell_0$. ■

An alternative method to show this equivalence is to reason about representation information, and show that the two representations are equivalent using ideas similar to [27], [38], [50].

THEOREM For all $s \in |\dagger counter|$,

$$\exists t_1 \in cell_0. t_1 \mapsto s \in \widehat{mkcounter}_1 \iff \exists t_2 \in cell_0. t_2 \mapsto s \in \widehat{mkcounter}_2$$

Proof: The fact that $\widehat{mkcounter}_1$ and $\widehat{mkcounter}_2$ are linear functions means that the t_1 and t_2 of the two existentials are unique whenever they exist.

For a trace $t \in cell_0$, say that t represents a state i if $t \cdot \langle \text{get}.i \rangle \in cell_0$. Define a relation $R \subseteq cell_0 \times cell_0$ by

$$t_1 R t_2 \iff \begin{aligned} &\exists i. t_1 \text{ represents the state } i \wedge \\ &\quad t_2 \text{ represents the state } -i \end{aligned}$$

Now, we strengthen the statement of the theorem as follows:

- If $t_1 \in cell_0$ such that $t_1 \mapsto s \in \widehat{mkcounter}_1$ then there exists $t_2 \in cell_0$ such that $t_2 \mapsto s \in \widehat{mkcounter}_2$ and $t_1 R t_2$.
- If $t_2 \in cell_0$ such that $t_2 \mapsto s \in \widehat{mkcounter}_2$ then there exists $t_1 \in cell_0$ such that $t_1 \mapsto s \in \widehat{mkcounter}_1$ and $t_1 R t_2$.

The strengthened hypothesis can be proved by induction on the length of s . The conclusion follows. \blacksquare

This proof illustrates how one can perform “state-based” reasoning though there are no explicit states in the model. Essentially, the idea is to express state information in terms of observable operations. It is also noteworthy that relational reasoning is available to us, even though no relation-preservation properties are explicitly postulated in the model.

The next example illustrates the “extensionality” properties of the semantics. The following equivalence is an example of an extensionality property for commands:

$$x := x + 1; x := x + 1 \quad \equiv \quad x := x + 2$$

One expects such equivalences to hold in the language of **while** programs. However, the equivalence fails for Algol. Suppose we substitute the following phrase for x :

$$\mathbf{if } y = 0 \mathbf{ then } y \mathbf{ else } z$$

where y and z are of type **var[int]**. The two sides of the equivalence give different results starting from an initial state where $y = 0$ and $z = 0$. We gather that variables in Algol are in general arbitrary objects with *get* and *put* operations. They are not necessarily what we called storage cells.

On the other hand, the **new** $[\delta]$ operation of Algol does create storage cells. So, one would expect the following variant of the equivalence to hold:

$$\begin{array}{l} \mathbf{new}[\mathbf{int}] \ x. \\ C_1; \\ p(x := x + 1; x := x + 1); \\ C_2 \end{array} \quad \equiv \quad \begin{array}{l} \mathbf{new}[\mathbf{int}] \ x. \\ C_1; \\ p(x := x + 2); \\ C_2 \end{array}$$

where C_1 and C_2 are arbitrary command phrases. This equivalence holds in the object-based model. It can be formally proved in the same fashion as the counter example using the relation

$$t_1 R t_2 \iff \exists i. t_1 \text{ and } t_2 \text{ represent the state } i$$

Thus, it may be gathered that the model is extensional in modelling commands even though it has an “intensional” feel.

The next few examples illustrate the “irreversible change” (or “single-threadedness”) properties of the model. (See discussion in [50].) Consider the equivalence:

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

where $f : \mathbf{exp[int]} \rightarrow \mathbf{exp[int]}$. This is essentially a “constant propagation” optimization. Surprisingly, the equivalence fails in most models of Algol. It is easy to understand this failure by noting that most models support the expression snap-back combinator. The function f could be $\lambda z. \mathbf{do } x := 1 \mathbf{ result } z$, in which case $f(x + y) = y + 1$ differs from $f(y) = y$. (Keep in mind that the parameter passing is by name.) This illustrates how the snap-back combinator destroys even mundane principles of reasoning for expressions.

The equivalence is trivial in the object-based model. Recall that the functions in $\mathbf{exp[int]} \Rightarrow \mathbf{exp[int]}$ are precisely the standard (stable or, in this case, continuous) functions $[int \rightarrow int]$. (Cf. Fact 5.) So, the standard reasoning for such functions continues to be valid.

As a final example, consider equivalence (2) from Sec. 1:

$$\begin{array}{l} \mathbf{new[int]} \ x. \\ \quad x := 0; \\ \quad p(x := x + 1); \\ \quad \mathbf{if } x > 0 \mathbf{ then undef else skip} \end{array} \quad \equiv \quad p(\mathbf{undef})$$

where $p : \mathbf{comm} \rightarrow \mathbf{comm}$. For the left hand side, if p runs its argument at all, then $x > 0$ holds after the call and divergence results. In this case, the right hand side diverges too. If p ignores its argument, $p(C)$ is independent of C . So, both the commands have the same behavior. As mentioned in connection with Fact 2, all known models of imperative programs fail to satisfy such equivalences because they model the snap-back combinator **try**. (Consider $p = \mathbf{try}$.)

On the other hand, the equivalence holds trivially in the object-based model. The meaning of both sides is $\{[p \rightarrow \langle \bar{0} \rangle] \mapsto *\}$.

Assertion-based reasoning

A widely used method for reasoning about imperative programs is in terms of assertions. Hoare logic [7] for **while** programs and Specification logic [65], [79] for higher-order procedures are canonical examples of such reasoning. In these logics, one reasons about commands using “Hoare triple” formulas $\{A\} C \{B\}$ where A and B are “assertions” and C is a command. The formula means that whenever the assertion A holds in a state, the execution of C yields a state in which the assertion B holds. Since the object-based semantic model does not mention states explicitly, it is not immediately clear how to interpret such formulas in the model. We briefly indicate how this may be done.

Recall, from Sec. 4.1, that the elements of an object space denote object behaviors as well as states. If $x \in \dagger X$ is an active element, every $s \in x$ yields an “ x -state” x/s . These notions can be generalized to interpretations of type contexts $[[\Pi \mid \Gamma]]$ in the obvious fashion. Let χ range over active elements of $[[\Pi \mid \Gamma]]$, η over trace environments and σ over χ -states of the form χ/η . We say that an assertion A holds in σ if there exists $\eta \in \sigma$ such that $\eta \mapsto tt \in [[A]]$. Then, the satisfaction of Hoare triples may be defined as follows:

Definition. If $\Pi \mid \Gamma \vdash \{A\}C\{B\} : \mathbf{formula}$ is a Hoare triple and $\chi \in \llbracket \Pi \mid \Gamma \rrbracket$ is an active element, χ is said to *satisfy* $\{A\}C\{B\}$ if, whenever A holds in a χ -state σ and there exists $\eta \in \sigma$ such that $\eta \mapsto * \in \llbracket C \rrbracket$, B holds in σ/η .

The assignment axiom for interference-controlled Algol programs is the following: Let $\Pi \mid \Gamma \vdash V : \mathbf{var}[\delta]$ and $\Pi \mid \Gamma \vdash E : \mathbf{exp}[\delta]$ be phrases. Then, for all assertion-valued functions $a : \mathbf{exp}[\delta] \rightarrow \mathbf{assert}$ (that are independent of V and E), we have

$$gv(V) \implies \{a(E)\} V := E \{a(\mathbf{deref} V)\}$$

where $gv(V)$ is a formula stating that V is a “good variable” [65], i.e., the regular function $\widehat{\llbracket V \rrbracket}$ gives \emptyset or $cell_i$ for some $i \in |\delta|$. It is easy to verify that the axiom is satisfied by all active elements $\chi \in \llbracket \Pi, a : \mathbf{var}[\delta] \rightarrow \mathbf{assert} \mid \Gamma \rrbracket$, showing that it is valid.¹⁶

8. Extensions

In this section, we briefly consider some extensions to the basic language considered in this paper.

Sum types

A type constructor for sum types $\theta_1 + \theta_2$ can be added with the usual syntax. Its interpretation is $\llbracket \theta_1 + \theta_2 \rrbracket = \dagger \llbracket \theta_1 \rrbracket + \dagger \llbracket \theta_2 \rrbracket$ where $+$ is the coproduct of coherent spaces (also called the “direct sum” [19], Sec. 12.1). This interpretation is in the same spirit as the lifted sum construction used with Scott domains and Girard’s “linearized” sum in [19], Sec. 12.5. The interpretation of terms is

$$\begin{aligned} \llbracket \Pi \mid \Gamma \vdash \mathbf{inl}(P) : \theta_1 + \theta_2 \rrbracket &= \{ \eta \mapsto 1.s : \eta \mapsto s \in \llbracket \Pi \mid \Gamma \vdash \widehat{P} : \theta_1 \rrbracket \} \\ \llbracket \Pi \mid \Gamma \vdash \mathbf{inr}(Q) : \theta_1 + \theta_2 \rrbracket &= \{ \eta \mapsto 2.t : \eta \mapsto t \in \llbracket \Pi \mid \Gamma \vdash \widehat{Q} : \theta_2 \rrbracket \} \\ \llbracket \Pi, \Pi' \mid \Gamma, \Gamma' \vdash \mathbf{case} P \mathbf{of} \mathbf{inl}(x_1) \Rightarrow Q_1 \mid \mathbf{inr}(x_2) \Rightarrow Q_2 : \theta' \rrbracket \\ &= \{ \eta \oplus \eta' \mapsto a : \\ &\quad \eta \mapsto i.s \in \llbracket \Pi \mid \Gamma \vdash P : \theta_1 + \theta_2 \rrbracket \wedge \\ &\quad \eta' \oplus [x_i \mapsto s] \mapsto a \in \llbracket \Pi' \mid \Gamma', x_i : \theta_i \vdash Q_i : \theta' \rrbracket \} \end{aligned}$$

The interpretation of **case** makes clear the reasons for the presence of \dagger in the interpretation of $\theta_1 + \theta_2$.

Independent product

The product type $\theta_1 \times \theta_2$ of interference-controlled Algol represents a composition of interfering components. One can add a separate type constructor for *independent*

products $\theta_1 \otimes \theta_2$ which puts together independent components. The term syntax for independent products [46] is given by:

$$\frac{\Pi_1 \mid \Gamma_1 \vdash P : \theta_1 \quad \Pi_2 \mid \Gamma_2 \vdash Q : \theta_2}{\Pi_1, \Pi_2 \mid \Gamma_1, \Gamma_2 \vdash P \otimes Q : \theta_1 \otimes \theta_2} \otimes \mathcal{I}$$

$$\frac{\Pi \mid \Gamma \vdash P : \theta_1 \otimes \theta_2}{\Pi \mid \Gamma \vdash \text{fst}(P) : \theta_1} \otimes 1\mathcal{E} \quad \frac{\Pi \mid \Gamma \vdash P : \theta_1 \otimes \theta_2}{\Pi \mid \Gamma \vdash \text{snd}(P) : \theta_2} \otimes 2\mathcal{E}$$

The independent product allows one to write “uncurried” procedures by using the isomorphism: $\theta_1 \otimes \theta_2 \rightarrow \theta' \cong \theta_1 \rightarrow (\theta_2 \rightarrow \theta')$. To give semantics to this extension, we need to extend the semantic framework as well. Essentially, we must enlarge the notion of “objects” to include composite objects that have multiple individual objects as components. Appendix A describes a framework of “finitary object spaces” which incorporates this extension. Another approach in terms of “dependence spaces” is described in [61].

Acceptors

In [66], Reynolds defines a type of *acceptors* corresponding to the l -values of variables. We postulate a primitive type $\mathbf{acc}[\delta]$ for acceptors (of δ -typed data values). The type of variables may then be defined as the product:

$$\mathbf{var}[\delta] = [\text{put} : \mathbf{acc}[\delta] \times \text{get} : \mathbf{exp}[\delta]]$$

The assignment operator $:=$ then has the (more general) type $\mathbf{acc}[\delta] \times \mathbf{exp}[\delta] \rightarrow \mathbf{comm}$, with an implicit coercion whenever a variable is used as the left hand side of $:=$.

To interpret the acceptor type, we define an active-passive coherent space $\mathbf{acc}[\delta]$ as follows:

$$|\mathbf{acc}[\delta]| = |\delta|, \quad \bigcirc_{\mathbf{acc}[\delta]} = \succ_{\delta}, \quad |\mathbf{acc}[\delta]|_{\wp} = \emptyset$$

Note that our original interpretation of variables satisfies $\mathbf{var}[\delta] = [\text{put} : \mathbf{acc}[\delta] \times \text{get} : \mathbf{exp}[\delta]]$. So, there is no substantial change to the semantics.

Reynolds also gives a treatment of $\mathbf{acc}[\delta]$ as the function space $\mathbf{exp}[\delta] \rightarrow \mathbf{comm}$. This treatment, however, is not useful for the interference-controlled language. We often want to apply the acceptor of a variable to an expression that involves the same variable, e.g., $x := x + 1$ desugared as $x.\text{put}(x.\text{get} + 1)$. Such phrases are not legal in interference-controlled Algol. On the other hand, we see that Reynolds’s conception is very well represented in our semantics as the isomorphism $\mathbf{acc}[\delta] \cong \delta \multimap \mathbf{comm}$.

Active expressions and monad combinators

Recent language design efforts in functional programming propose to add state-manipulation facilities to functional languages using monad combinators [42], [52],

[53]. While it is not entirely clear how to provide interference control for such combinators, we can certainly consider a simple form of monad types where interference control issues do not arise.

Let $\mathbf{act}[\delta]$ denote the type of computations (*active expressions*) that may change state and eventually return data values of type δ . The basic combinators are:

$$\frac{\Pi \mid \Gamma \vdash E : \mathbf{exp}[\delta]}{\Pi \mid \Gamma \vdash \mathbf{return} E : \mathbf{act}[\delta]} \quad \frac{\Pi \mid \Gamma \vdash P : \mathbf{act}[\delta] \quad \Pi \mid \Gamma, x : \mathbf{exp}[\delta] \vdash Q : \mathbf{act}[\delta']}{\Pi \mid \Gamma \vdash P \triangleright x. Q : \mathbf{act}[\delta']}$$

It is also useful to extend the command sequencing combinator “;” for active expressions. All these combinators can be treated as constants:

$$\begin{aligned} \mathbf{return} & : \mathbf{exp}[\delta] \rightarrow_p \mathbf{act}[\delta] \\ \mathbf{-} \triangleright \mathbf{-} & : \mathbf{act}[\delta] \times (\mathbf{exp}[\delta] \rightarrow \mathbf{act}[\delta']) \rightarrow_p \mathbf{act}[\delta'] \\ \mathbf{-}; \mathbf{-} & : \mathbf{comm} \times \mathbf{act}[\delta] \rightarrow_p \mathbf{act}[\delta] \end{aligned}$$

A simple example of these combinators is the accumulator object (Cf. Sec. 4.2):

$$\begin{aligned} \mathbf{accum} & = \mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{act}[\mathbf{int}] \\ \mathbf{mkaccum} & : \mathbf{var}[\mathbf{int}] \rightarrow_p \mathbf{accum} \\ \mathbf{mkaccum}(v) & = \lambda i. v := v + i; \mathbf{return} (\mathbf{deref} v) \end{aligned}$$

To give semantics to active expressions, we define an active-passive space $\mathbf{act}[\delta]$ as follows:

$$|\mathbf{act}[\delta]| = |\delta|, \quad \bigcirc_{\mathbf{act}[\delta]} = \bigcirc_{\delta}, \quad |\mathbf{act}[\delta]|_{\wp} = \emptyset$$

This is similar to $\mathbf{exp}[\delta]$ except that it has no passive tokens. The meanings of the above combinators are then given by the linear maps:

$$\begin{aligned} \mathbf{return}_{\delta} & : \mathbf{exp}[\delta] \rightarrow \mathbf{act}[\delta] \\ & = \{ i \mapsto i : i \in |\delta| \} \\ \mathbf{bind}_{\delta, \delta'} & : \mathbf{act}[\delta] \times (\mathbf{exp}[\delta] \Rightarrow \mathbf{act}[\delta']) \rightarrow \mathbf{act}[\delta] \\ & = \{ \langle \mathbf{fst}.i, \mathbf{snd}.(p \mapsto j) \rangle \mapsto j : i \in |\delta| \wedge j \in |\delta'| \wedge p \subseteq \{i\} \} \\ \mathbf{seq}_{\delta} & : \mathbf{comm} \times \mathbf{act}[\delta] \rightarrow \mathbf{act}[\delta] \\ & = \{ \langle \mathbf{fst}.*, \mathbf{snd}.i \rangle \mapsto i : i \in |\delta| \} \end{aligned}$$

Promotion for active expressions

Recent functional languages with state [52], [42], [75] also contain proposals for promotion of active expressions. Such expressions allocate and use state variables locally but do not have any effects on global variables. Thus, they may be viewed as “applicative” expressions from the outside.

To provide this feature, we add a promotion construct for active expressions by the type rule:

$$\frac{\Pi \mid \vdash P : \mathbf{act}[\delta]}{\Pi \mid \vdash \mathbf{run}(P) : \mathbf{exp}[\delta]} \textit{Promote}_\delta$$

Note the similarity with the promotion rule for passive functions $\rightarrow_p \mathcal{E}$.

The semantics of the promotion construct, as in Sec. 5, makes no change to the meaning of the phrase:

$$\llbracket \Pi \mid \vdash \mathbf{run}(P) : \mathbf{exp}[\delta] \rrbracket = \llbracket \Pi \mid \vdash P : \mathbf{act}[\delta] \rrbracket$$

This interpretation is sound because the active type context of P is empty (and the meaning is a passivity-reflecting function).

Similar promotion facilities have been used in traditional Algol-like languages (without active expressions). In this context, they are called “block expressions” [79], Sec. 9.7, but the essential idea is similar.

9. Conclusion

We 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.

The main difference between our semantics and the conventional approaches is that we model objects by their observable behavior without reference to their internal states. This decision obtains theoretical economy and allows us to formulate a ground-level domain-theoretic model of object behaviors. In contrast, conventional approaches use explicit states and model objects as state transformers. But, to represent local variable abstractions correctly, they must characterize object functions as being “uniform” in the underlying state sets.

The functor category approaches [51], [66], [78] attempt to model uniformity by natural transformations. This approach has not yet led to satisfactory solutions because the functors involved have “mixed variance” and naturality is not enough to capture uniformity. In contrast, the relational parametricity models [50], [72] model uniformity by preservation of relations. The recent full abstraction result of Sieber [73] suggests that this approach can be quite successful.

Stepping back, we can see that the idea of objects with local states is at the heart of the above approaches as well (though it may not have been explicitly stated in those terms). The main difference is then the treatment of state sets. By eschewing the explicit treatment of states, we are able to avoid second-order concepts like functors and finesse the issue of uniformity. On the other hand, we believe that both the explicit state view and the behavior-based views have useful roles to play. The insights obtained in the current study should prove useful for resolving the outstanding issues in the explicit state view.

The most important of these issues is modelling the irreversibility of state changes. All known explicit state models (for higher-order languages) contain the snap back

operator **try**. All but Tennent’s model [78] also contain the more troublesome **do-result** operator (which invalidates common reasoning for expressions). The present model is the first to avoid such state change reversals. Note, however, that Proposition 4 already gives an indication of how irreversible state changes can be modelled in an explicit state view. This should be explored further in the future work.

Our model also brings to light important connections between the semantics of state and other subjects in programming language theory. One of them, viz., connection to linear logic, is studied in [60]. This should pave the way for good operational models of stateful languages such as geometry of interaction [3], [18], games [5], [29] and sequential algorithms [14]. A second connection, to parametricity, arises in studying the relationship to the explicit state model. At least for the types that arise in the semantics of interference-controlled Algol, our approach shows that the independence of a parametric type on its type parameters can be explicitly modelled (via a suitable generalization of Proposition 4). It is an open question whether this can be carried further for more general parametric types. A third connection is to concurrency theory. Note that the behavior-based implicit state view taken here is a common practice in concurrency [28], [39]. The use of simulations as morphisms is also found in recent models of concurrency [13], [22]. These connections should prove useful for studying concurrent imperative languages.

Finally, an important issue that is left untouched in this work is the development of uniform reasoning principles. In a sense, the traditional methods for reasoning, including assertion-based methods and relational parametricity-based methods, can be said to reflect an explicit state view of the semantics. What reasoning methods would be contributed by the behavior-based view taken here?

Acknowledgements

This work owes much to the encouragement and inspiration provided by Peter O’Hearn throughout its development. The feedback from him, Bob Tennent and Phil Wadler were crucial for the development of the current presentation. Anonymous referees pointed out many a rough spot. Special thanks go to William Harrison and Howard Huang for their help in proof-reading.

Thanks to Christian Retoré for providing the foundational ideas of pomset logic (which are implicit throughout this work) and Samson Abramsky for pointing me in this direction. I thank Guo-Qiang Zhang for explaining his constructions for dI-domains (used in finitary object spaces - Appendix A).

This research was supported by National Science Foundation under grant NSF-CCR-93-03043.

Appendix A

Categorical Structure of the Model

The model of interference-controlled Algol presented here is an instance of a general construction outlined in [9], Theorem 8. (See also [11].) We use this construction to explain the categorical structure of the model.

The body of the paper mentions two base categories: the category of coherent spaces and linear maps **CohL** in Section 4, and the category of active-passive coherent spaces and (passivity-preserving) linear maps **APCohL** in Section 5. Both the categories are *symmetric monoidal closed* with tensor products given by \otimes , the tensor unit $\mathbf{1}$ and the exponential given by \multimap . Let us factor out the commonalities by assuming a symmetric monoidal closed category $(\mathbf{C}, \otimes, \mathbf{1}, \multimap)$ with finite products (\times, \top) .

First, we define a symmetric monoidal functor $\dagger : \mathbf{C} \rightarrow \mathbf{C}$ and extend it to a comonad by defining monoidal natural transformations:

$$\begin{aligned} \text{read}_A & : \dagger A \rightarrow A &= \{ \langle a \rangle \mapsto a : a \in |A| \} \\ \text{dup}_A & : \dagger A \rightarrow \dagger\dagger A &= \{ s_1 \cdots s_n \mapsto \langle s_1, \dots, s_n \rangle : s_1 \cdots s_n \in |\dagger A| \} \end{aligned}$$

to say that \dagger is a “symmetric monoidal” functor is to say that there are natural transformations:

$$\begin{aligned} \text{merge}_{A,B} & : \dagger A \otimes \dagger B \rightarrow \dagger(A \otimes B) \\ &= \{ (\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle) \mapsto \langle (a_1, b_1), \dots, (a_n, b_n) \rangle \\ &\quad : a_1, \dots, a_n \in |A| \wedge b_1, \dots, b_n \in |B| \} \\ \text{merge}_{\mathbf{1}} & : \mathbf{1} \rightarrow \dagger\mathbf{1} \\ &= \{ * \mapsto \langle * \rangle^n : n \geq 0 \} \end{aligned}$$

which commute with the symmetric monoidal structure in an appropriate fashion [9]. To say that **read** and **dup** are monoidal transformations is to imply that they commute with the **merge** transformations.

Next, we consider the (Eilenberg-Moore) category \mathbf{C}^\dagger of \dagger -coalgebras and coalgebra morphisms [34]. Recall that a *coalgebra* is a pair $\langle C, h_C : C \rightarrow \dagger C \rangle$ of an object C and an arrow $h_C : C \rightarrow \dagger C$ such that the requisite diagrams commute. The object C is called the underlying object of the coalgebra and h_C its structure map. We often write the coalgebra as simply C with the structure map understood. Coalgebra morphisms $f : C \rightarrow D$ are arrows $f : C \rightarrow D$ of the underlying objects such that the following square commutes:

$$\begin{array}{ccc} C & \xrightarrow{f} & D \\ h_C \downarrow & & \downarrow h_D \\ \dagger C & \xrightarrow{\dagger f} & \dagger D \end{array}$$

What we call “object spaces” are (special classes of) \dagger -coalgebras and “regular functions” are coalgebra morphisms.

A free coalgebra is a pair $\langle \dagger A, \text{dup}_A : \dagger A \rightarrow \dagger\dagger A \rangle$ for an object A of \mathbf{C} . Notice the (natural) isomorphism

$$\mathbf{C}^\dagger(C, \dagger A) \cong \mathbf{C}(UC, A)$$

where $U : \mathbf{C}^\dagger \rightarrow \mathbf{C}$ is the “underlying object functor”. The maps in the two directions are:

$$\begin{aligned} \hat{f} &= C \xrightarrow{h_C} \dagger C \xrightarrow{\dagger f} \dagger A \quad \text{for any } f : UC \rightarrow A \\ \check{g} &= UC \xrightarrow{Ug} \dagger A \xrightarrow{\text{read}_A} A \quad \text{for any } g : C \rightarrow \dagger A \end{aligned}$$

Morphisms of the form \hat{f} are called “regular extensions” of linear maps in Section 4. Likewise, \check{g} is the “linear pattern” of a regular function. The semantics of interference-controlled Algol is given using free coalgebras.

If C and D are coalgebras, their tensor product $C \otimes D$ can be made into a coalgebra by associating the structure map:

$$C \otimes D \xrightarrow{h_C \otimes h_D} \dagger C \otimes \dagger D \xrightarrow{\text{merge}_{C,D}} \dagger(C \otimes D)$$

The tensor unit is $\langle \mathbf{1}, \text{merge}_1 : \mathbf{1} \rightarrow \dagger\mathbf{1} \rangle$. This gives a symmetric monoidal structure to \mathbf{C}^\dagger . However, $C \otimes D$ is not a free coalgebra even if C and D are. So, we cannot simply work in the subcategory of free coalgebras. This is the reason for using a multicategory structure in Section 4. We require, however, that $\langle \mathbf{1}, \text{merge}_1 \rangle$ is the terminal object in \mathbf{C}^\dagger . This gives a unique coalgebra morphism

$$\text{discard}_C : C \rightarrow_R \mathbf{1}$$

which is used for interpreting “weakening” (implicit in the rule *Id*).

For any coalgebra C and a free coalgebra $\dagger A$, there exists an exponent $C \Rightarrow \dagger A = \dagger(UC \multimap A)$. This is clear from the isomorphisms

$$\mathbf{C}^\dagger(D \otimes C, \dagger A) \cong \mathbf{C}(UD \otimes UC, A) \cong \mathbf{C}(UD, UC \multimap A) \cong \mathbf{C}^\dagger(D, \dagger(UC \multimap A))$$

The associated combinators are:

$$\begin{aligned} \Lambda(f) &= D \xrightarrow{h_D} \dagger D \xrightarrow{\dagger \Lambda(\text{read}_A \circ f)} \dagger(C \multimap A) \\ \text{apply}_{C, \dagger A} &= \dagger(C \multimap A) \otimes C \xrightarrow{h} \dagger(\dagger(C \multimap A) \otimes C) \\ &\quad \xrightarrow{\dagger(\text{read} \otimes \text{id})} \dagger((C \multimap A) \otimes C) \xrightarrow{\dagger \text{apply}} \dagger A \end{aligned}$$

Thus, in the words of [9], “the full subcategory of finite tensor products of free coalgebras forms a symmetric monoidal closed category” whenever \mathbf{C} is symmetric monoidal closed and \dagger is a symmetric monoidal comonad.

If \mathbf{C} has products (which is the case for our base categories), the free coalgebras have products. The product of $\dagger A$ and $\dagger B$ is $\dagger(A \times B)$. The maps involved in this structure are:

- $\dagger(A \times B) \xrightarrow{\dagger\pi_1} \dagger A$ and $\dagger(A \times B) \xrightarrow{\dagger\pi_2} \dagger B$.
- $C \xrightarrow{h_C} \dagger C \xrightarrow{\dagger(\tilde{f}, \tilde{g})} \dagger(A \times B)$.

Recall that $\mathbf{1}$ is already the terminal object and is now seen to be isomorphic to $\dagger\mathbb{T}$.

It is easily verified that, by considering arrows $f : \dagger A_1 \otimes \cdots \otimes \dagger A_n \rightarrow B$ of \mathbf{C} as multi-arrows $f : A_1, \dots, A_n \rightarrow B$, we can form a multicategory. We call this the “Kleisli multicategory” of \mathbf{C} under \dagger and denote it by $\mathbf{C}_{\dagger\otimes}$. The semantics of Section 4.3 is given in such a multicategory.

All of this captures the surface structure of the model, given an appropriate comonad \dagger . It does not, however, explain \dagger (which is where the novelty of the present model lies). An earlier paper [59] defines \dagger as the free comonoid comonad with respect to an additional (non-symmetric) monoidal structure $(\triangleright, \mathbf{1})$. The bifunctor \triangleright called “before” represents a notion of sequential composition. See [63] for a proof-theoretical study of this bifunctor.

Passivity

The category of active-passive coherent spaces used for modelling passivity has important elements which can also be expressed at the categorical level.

Our prototypical base category \mathbf{C} (such as **APCohL**) has a full subcategory \mathbf{P} for modelling passive types. This subcategory is reflective as well as co-reflective. (See [34], Sec. IV.3.) This means that there are functors $\wp, \mathcal{P} : \mathbf{C} \rightarrow \mathbf{C}$ taking values in \mathbf{P} such that we have the natural isomorphisms:

$$\begin{aligned} \mathbf{passify}^{-1} : \mathbf{P}(\wp A, P) &\cong \mathbf{C}(A, P) : \mathbf{passify} \\ \mathbf{promote}^{-1} : \mathbf{P}(P, \mathcal{P} B) &\cong \mathbf{C}(P, B) : \mathbf{promote} \end{aligned}$$

By the first isomorphism, we obtain a natural transformation $\mathbf{pas}_A : A \rightarrow \wp A$ as the unit of the reflection, and, by the second isomorphism, $\mathbf{der}_B : \mathcal{P} B \rightarrow B$ as the co-unit of the co-reflection.

For active-passive coherent spaces, the functor \wp assigns, to each space A , its passive subspace $\wp A$ and, to each (passivity-reflecting) linear map $f : A \rightarrow_L B$, the corresponding restriction $\wp f : \wp A \rightarrow_L \wp B$. The unit \mathbf{pas}_A is given by $\{a \mapsto a : a \in |A|_{\wp}\}$. The isomorphism **passify** and the unit **pas** are used in interpreting the type rules *Passify* and *Activate* respectively.

The functor \mathcal{P} assigns to each space A , the passive space $\mathcal{P} A$ with $|\mathcal{P} A| = |\mathcal{P} A|_{\wp} = |A|$ and the same coherence relation as A . A (passivity-reflecting) linear map $f : A \rightarrow_L B$ is mapped to the same map regarded as an arrow $\mathcal{P} A \rightarrow_L \mathcal{P} B$. The co-unit \mathbf{der}_B is given by $\{b \mapsto b : b \in |B|\}$. The isomorphism **promote** and the co-unit **der** are used in interpreting the type rules *Promote* and *Derelict* respectively.

Note that the functor \wp is full. This makes the unit \mathbf{pas}_A a split epi. The right inverse of \mathbf{pas}_A is then a monic, $\mathbf{act}_A : \wp A \rightarrow A$. We may thus call $\wp A$ the “passive subobject” of A (terminology first used in [45]).

The full subcategory \mathbf{P} is closed under all the constructions mentioned previously: \otimes , $\mathbf{1}$, $A \multimap (-)$, \times , \top , and \dagger . Further, the reflector \wp preserves all of this structure on the nose:

$$\begin{aligned} \wp \mathbf{1} &= \mathbf{1} & \wp(A \otimes B) &= \wp A \otimes \wp B \\ \wp \dagger A &= \dagger \wp A & \wp(A \multimap B) &= A \multimap \wp B \\ \wp \top &= \top & \wp(A \times B) &= \wp A \times \wp B \end{aligned}$$

Since \wp preserves the monoidal structure, the associated reflection is a monoidal adjunction. This fact is also used in the interpretation of *Passify* and *Activate*.

Finally, the passive sublanguage of interference-controlled Algol is a functional programming language. Therefore, the tensor product in $\mathbf{P}_{\dagger \otimes \dagger}$ must be the categorical product. One way of ensuring this is to notice that free \dagger coalgebras in \mathbf{P} have “diagonals”, i.e., there is a monoidal natural family of coalgebra morphisms:

$$\text{dup}_P : \dagger P \rightarrow_R \dagger P \otimes \dagger P = \{ p_1 \cup p_2 \mapsto (p_1, p_2) : p_1 \cup p_2 \in |\dagger P| \}$$

such that $\langle P, \text{dup}_P, \text{discard}_P \rangle$ forms a commutative comonoid. Further, all coalgebra morphisms are comonoid morphisms. It then follows that tensor products in $\mathbf{P}_{\dagger \otimes \dagger}$ are isomorphic to products by [9], Theorem 9. In other words, we have an isomorphism

$$\dagger P \otimes \dagger Q \cong \dagger(P \times Q)$$

given by

$$\text{pack}_{P,Q} : \dagger P \otimes \dagger Q \rightarrow \dagger(P \times Q) = \{ (p \downarrow 1, p \downarrow 2) \mapsto p : p \in |\dagger(P \times Q)| \}$$

where $p \downarrow 1 = \{ \text{fst}.a : a \in p \}$ and $p \downarrow 2 = \{ \text{snd}.b : b \in p \}$.

Note that the Kleisli multicategory $\mathbf{P}_{\dagger \otimes \dagger}$ is equivalent to the Kleisli category \mathbf{P}_{\dagger} because multicategory maps $\dagger P_1 \otimes \cdots \otimes \dagger P_n \rightarrow Q$ are one-to-one with maps $\dagger(P_1 \times \cdots \times P_n) \rightarrow Q$. The Kleisli category is cartesian closed.

Finitary object spaces

The multicategory structure outlined above is not entirely ideal. For certain applications, such as the Yoneda embedding used for modelling interference [47], one would like to use an ordinary category rather than a multicategory. The difficulty is that the free coalgebras do not have tensor products while the category of finite tensor products of free coalgebras does not have products. So, some other approach is needed. There seem to be two alternatives.

One can enrich coherent spaces with further structure so that free coalgebras over such enriched coherent spaces are closed under tensor product. Structures called *dependence spaces* are described in [61] with this property. In addition to having tensor products, these structures also give a smooth treatment of passivity.

The second alternative is to find a class of coalgebras that are closed under tensor product. We explore this alternative here by defining a class of coalgebras called *finitary object spaces*. The main issue in formulating object spaces other than the free ones is to calculate the exponentials. While the exponents of free coalgebras can be represented by free coalgebras, the more general exponents cannot be so represented. So, we need to examine the structure of morphisms more closely.

We define an *object space* as a coherent space $(|C|, \circ_C)$ where the set of tokens is equipped with the structure of a partial monoid $|C| = (|C|, \cdot, e_C)$ such that the following condition is satisfied:

$$x_1 \cdot y_1 \circ x_2 \cdot y_2 \implies (x_1 \circ x_2 \wedge (x_1 = x_2 \implies y_1 = y_2))$$

To say that $|C|$ is a *partial monoid* is to say that its multiplication operation “ \cdot ” is a partial function. (We often omit “ \cdot ” in writing a product xy .) Note that an object space is a \dagger -coalgebra with the structure map:

$$h_C = \{ x_1 \cdots x_n \mapsto \langle x_1, \dots, x_n \rangle : n \geq 0 \wedge x_1, \dots, x_n \in |C| \wedge x_1 \cdots x_n \in |C| \}$$

There exists a preorder on the monoid of tokens defined by $x \preceq y \iff \exists x'. xx' = y$. Note that, if a product xy is defined, then xy' is defined for all $y' \preceq y$. Secondly, $x \preceq y$ implies $x' \circ y'$ for all $x' \preceq x, y' \preceq y$. In particular, $e_C \circ x$ for all $x \in |C|$. We call an object space *finitary* if \preceq is a finitary partial order. (The set $\{y : y \preceq x\}$ is finite for all $x \in |C|$.) *Regular maps* $f : C \rightarrow_R D$ are coalgebra morphisms, as usual. Thus, we form a full subcategory **FOb** of **CohL** † .

A free object space $\dagger A$ is clearly a finitary object space. Its finitary partial order is the prefix order. It is easy to see that finitary object spaces carry the structure of prime event structures which, in turn, correspond to dI-domains [83]. Regular maps are special cases of linear maps on dI-domains.

The tensor product $C \otimes D$ of object spaces is the tensor product of the underlying coherent spaces with the monoid structure being the monoid product $|C| \times |D|$. The object space **1** has the one-element monoid structure. Note that **1** is the terminal object of **FOb**. The product $C \times D$ of finitary object spaces has the token-monoid $|C \times D| = |C| \oplus |D|$, the monoid coproduct of $|C|$ and $|D|$.

To define the exponent $C \Rightarrow D$ of finitary object spaces, we need to consider another class of maps called active maps. An *active map* $f : C \rightarrow_A D$ is a linear map such that

1. $e_C \mapsto e_D \in f$, and
2. $x \mapsto y \in f$ and $y' \preceq y$ implies there exists (unique) $x' \preceq x$ such that $x' \mapsto y' \in f$.

Note that active maps properly include regular maps. (They still form a subclass of linear maps on dI-domains.) The tokens of $C \Rightarrow D$ are active maps with a maximum pair $x \mapsto y$. Call them *prime active maps*. For an active map $f : C \rightarrow_A D$ and pair $x \mapsto y \in f$, let $[x \mapsto y]_f$ denote the set

$$[x \mapsto y]_f = \{ x' \mapsto y' \in f : x' \preceq x \wedge y' \preceq y \}$$

Then all prime active maps can be written as $[x \mapsto y]_f$ for some $f : C \rightarrow_A D$ and $x \mapsto y \in f$. The exponent $C \Rightarrow D$ is defined as follows:

$$\begin{aligned} |C \Rightarrow D| &= \{ [x \mapsto y]_f : f : C \rightarrow_A D \wedge x \mapsto y \in f \} \\ [x_1 \mapsto y_1]_{f_1} \circ [x_2 \mapsto y_2]_{f_2} &\iff \exists f : C \rightarrow_A D. [x_1 \mapsto y_1]_{f_1} \cup [x_2 \mapsto y_2]_{f_2} \subseteq f \end{aligned}$$

with the monoid structure:

$$\begin{aligned} e_{C \Rightarrow D} &= \{ e_C \mapsto e_D \} \\ [x_1 \mapsto y_1]_{f_1} \cdot [x_2 \mapsto y_2]_{f_2} &= [x_1 x_2 \mapsto y_1 y_2]_f \\ &\quad \text{where } f = [x_1 \mapsto y_1]_{f_1} \cup \{ x_1 x' \mapsto y_1 y' : x' \mapsto y' \in [x_2 \mapsto y_2]_{f_2} \} \end{aligned}$$

(Note that the above product of prime active maps is defined if and only if the products $x_1 x_2$ and $y_1 y_2$ are both defined.) This construction closely parallels the construction of exponents for dI-domains [86]. To see that $C \Rightarrow D$ is indeed the exponent in **FOb**, suppose $f : B \otimes C \rightarrow_R D$ is a regular map. For any $x \in |B|$, we can define an active map $f/x : C \rightarrow_A D$ by

$$f/x = \{ y' \mapsto z' : \exists x' \leq x. (x', y') \mapsto z' \in f \}$$

Then, the natural isomorphism $B \otimes C \rightarrow_R D \cong B \rightarrow_R (C \Rightarrow D)$ is given by the combinators:

$$\begin{aligned} \Lambda(f) &: B \rightarrow_R (C \Rightarrow D) \\ &= \{ x \mapsto [y \mapsto z]_{f/x} : (x, y) \mapsto z \in f \} \\ \text{apply}_{C,D} &: (C \Rightarrow D) \otimes C \rightarrow_R D \\ &= \{ ([y \mapsto z]_f, y) \mapsto z : f : C \rightarrow_A D \wedge y \mapsto z \in f \} \end{aligned}$$

Lengthy calculations show that these maps are regular and give the required isomorphism.

To define active-passive object spaces **APOb**, we equip object spaces $(|C|, \circ_C)$ with designated submonoids $|C|_\wp$ such that, for all $x, y \in |C|_\wp$,

$$\begin{aligned} xy &= yx \\ x \preceq y &\implies xy = y \end{aligned}$$

(Since $xy \circ yx$, the product xy of passive tokens is defined only if $x \circ y$.)

To summarize, we have the following hierarchy of full subcategories:

$$\begin{aligned} \mathbf{CohL}_\dagger &\subseteq \mathbf{CohL}_{\dagger \otimes \dagger} \subseteq \mathbf{FOb} \subseteq \mathbf{Ob} \subseteq \mathbf{CohL}^\dagger \\ \mathbf{APCohL}_\dagger &\subseteq \mathbf{APCohL}_{\dagger \otimes \dagger} \subseteq \mathbf{APFOb} \subseteq \mathbf{APOb} \subseteq \mathbf{APCohL}^\dagger \end{aligned}$$

where the notation $\mathbf{C}_{\dagger \otimes \dagger}$ is used for the category of finite tensor products of finite coalgebras. **FOb** and **APFOb** are the least categories in these hierarchies which are closed under all of tensor products, exponentials and products. The semantics of interference-controlled Algol can be viewed as working in the category **APFOb**.

Appendix B

Proof of Operational Adequacy

The proof follows the general plan of Plotkin [56]. Since this is standard, we mainly focus on the parts of the proof related to the handling of state.

We call a phrase P *semi-closed* if all its free identifiers are of types $\mathbf{var}[\delta]$. We use Δ to range over variable type contexts. A semi-closed phrase of a *base type* ($\mathbf{exp}[\delta]$ or \mathbf{comm}) is said to be a *normal phrase* if its only redexes are instances of \mathbf{undef} . So, normal phrases satisfy the following context-free syntax:

$$\begin{aligned} \text{(expressions)} \quad E &::= 0 \mid E_1 + E_2 \mid \mathbf{deref} \ x \mid \mathbf{if}(E, E_1, E_2) \mid \mathbf{undef} \\ \text{(commands)} \quad C &::= \mathbf{skip} \mid C_1; C_2 \mid C_1 \parallel C_2 \mid x := E \mid \mathbf{if}(E, C_1, C_2) \\ &\quad \mid \mathbf{new}[\delta] \ x. C \mid \mathbf{undef} \end{aligned}$$

If P is a semi-closed phrase of a base type, $\omega(P)$ denotes the phrase obtained by replacing all its redexes by \mathbf{undef} . Note that $\llbracket \omega(P) \rrbracket \subseteq \llbracket P \rrbracket$.

To relate states involved in the operational semantics and traces involved in the denotational one, we define some relations. For $s \in |\dagger \mathit{var}[\delta]|$, define a relation $\xrightarrow{s} \subseteq |\delta| \times |\delta|$ inductively by

$$\begin{aligned} i &\xrightarrow{\langle \rangle} i' \iff i = i' \\ i &\xrightarrow{\langle \mathit{get}.j \rangle \cdot s} i' \iff i = j \wedge i \xrightarrow{s} i' \\ i &\xrightarrow{\langle \mathit{put}.j \rangle \cdot s} i' \iff j \xrightarrow{s} i' \end{aligned}$$

We extend this to states and trace environments (with respect to a context $\Delta \mid \Delta'$) by

$$\sigma \xrightarrow{\eta} \sigma' \iff \sigma(x) \xrightarrow{\eta(x)} \sigma'(x) \text{ for all identifiers } x \text{ in } \Delta \mid \Delta'$$

Note the following facts:

1. The relation $\xrightarrow{\eta}$ is single-valued, i.e., $\sigma \xrightarrow{\eta} \sigma'$ and $\sigma \xrightarrow{\eta} \sigma''$ implies $\sigma' = \sigma''$.
2. If η is a passive trace environment, $\sigma \xrightarrow{\eta} \sigma'$ implies $\sigma = \sigma'$.
3. $\sigma \xrightarrow{\eta} \sigma'$ for some σ and σ' if and only if each $\eta(x)$ is a storage cell trace.
4. If $\sigma \xrightarrow{\eta_1} \sigma_1$ and $\sigma \xrightarrow{\eta_2} \sigma_2$ then $\eta_1 \circ \eta_2$.

LEMMA 1 *The semantics of normal phrases is sound and adequate for execution, i.e., for any normal phrase $\Delta \mid \Delta' \vdash P : \theta$,*

- if θ is $\mathbf{exp}[\delta]$ then $(\sigma, P) \Downarrow i \iff \exists \eta \in \llbracket \Delta \mid \Delta' \rrbracket. \sigma \xrightarrow{\eta} \sigma \wedge \eta \mapsto i \in \llbracket P \rrbracket$,
- if θ is \mathbf{comm} then $(\sigma, P) \Downarrow \sigma' \iff \exists \eta \in \llbracket \Delta \mid \Delta' \rrbracket. \sigma \xrightarrow{\eta} \sigma' \wedge \eta \mapsto * \in \llbracket P \rrbracket$.

Proof: By induction on the structure of P . (Note that the η 's in the existentials are unique.) This is a straightforward verification. We show a few cases:

Suppose $P = E_1 + E_2$. By definition of \Downarrow , $(\sigma, P) \Downarrow i$ iff there exist i_1, i_2 such that $i = i_1 + i_2$, $(\sigma, E_1) \Downarrow i_1$ and $(\sigma, E_2) \Downarrow i_2$. By the inductive hypothesis, this is equivalent to the existence of η_1, η_2 such that $\sigma \xrightarrow{\eta_1} \sigma, \sigma \xrightarrow{\eta_2} \sigma, \eta_1 \mapsto i_1 \in \llbracket E_1 \rrbracket$ and $\eta_2 \mapsto i_2 \in \llbracket E_2 \rrbracket$. All this amounts to the existence of $\eta = \eta_1 \cdot \eta_2$ such that $\sigma \xrightarrow{\eta} \sigma$ and $\eta \mapsto i \in \llbracket E_1 + E_2 \rrbracket$.

Suppose $P = \mathbf{deref} \ x$. Then $(\sigma, P) \Downarrow i$ iff $i = \sigma(x)$. Let $\eta = \eta_0[x \mapsto \langle \text{get}.i \rangle]$. We have $\sigma \xrightarrow{\eta} \sigma$ and $\eta \mapsto i \in \llbracket \mathbf{deref} \ x \rrbracket$. Since η is uniquely determined, the reverse implication holds as well.

Suppose $P = (x := E)$. Then $(\sigma, P) \Downarrow \sigma'$ iff there exists i such that $(\sigma, E) \Downarrow i$ and $\sigma' = \sigma[x \mapsto i]$. By the inductive hypothesis, $(\sigma, E) \Downarrow i$ is equivalent to the existence of η_1 such that $\sigma \xrightarrow{\eta_1} \sigma$ and $\eta_1 \mapsto i \in \llbracket E \rrbracket$. Let $\eta_2 = \eta_0[x \mapsto \langle \text{put}.i \rangle]$. We have $\sigma \xrightarrow{\eta_1 \cdot \eta_2} \sigma'$ and $\eta_1 \cdot \eta_2 \mapsto * \in \llbracket x := E \rrbracket$.

Suppose $P = C_1 \parallel C_2$. Then $(\sigma, P) \Downarrow \sigma'$ iff there exist corresponding decompositions $\sigma = \sigma_1 \oplus \sigma_2$ and $\sigma' = \sigma'_1 \oplus \sigma'_2$ such that $(\sigma_1, C_1) \Downarrow \sigma'_1$ and $(\sigma_2, C_2) \Downarrow \sigma'_2$. By the inductive hypothesis, this is equivalent to the existence of η_1, η_2 such that $\sigma_1 \xrightarrow{\eta_1} \sigma'_1, \sigma_2 \xrightarrow{\eta_2} \sigma'_2, \eta_1 \mapsto * \in \llbracket C_1 \rrbracket$ and $\eta_2 \mapsto * \in \llbracket C_2 \rrbracket$. We have $\sigma_1 \oplus \sigma_2 \xrightarrow{\eta_1 \oplus \eta_2} \sigma'_1 \oplus \sigma'_2$ and $\eta_1 \oplus \eta_2 \mapsto * \in \llbracket C_1 \parallel C_2 \rrbracket$.

Suppose $P = \mathbf{new}[\delta] \ x.C$. Then $(\sigma, P) \Downarrow \sigma'$ iff there exists i such that $(\sigma \oplus [x \mapsto \text{init}[\delta]], C) \Downarrow \sigma' \oplus [x \mapsto i]$. By the inductive hypothesis, this is equivalent to the existence of η and s such that $\sigma \oplus [x \mapsto \text{init}[\delta]] \xrightarrow{\eta \oplus [x \mapsto s]} \sigma' \oplus [x \mapsto i]$ and $\eta \oplus [x \mapsto s] \mapsto * \in \llbracket C \rrbracket$. So, we have $\sigma \xrightarrow{\eta} \sigma'$ and, since $s \in \text{cell}_{\text{init}[\delta]}$, $\eta \mapsto * \in \llbracket \mathbf{new}[\delta] \ x.C \rrbracket$. ■

LEMMA 2 *Semantics is sound for reduction, i.e., if $P \longrightarrow^* P'$ then $\llbracket P \rrbracket = \llbracket P' \rrbracket$.*

Proof: By induction on the length of the reduction sequence $P \longrightarrow^* P'$ and the structure of P . It only remains to be seen that each reduction rule is semantics-preserving. This is a straightforward verification. As an example, we show the property for the last propagation rule:

$$\mathbf{if}(P_0, P_1, P_2) := Q \longrightarrow \mathbf{if}(P_0, P_1 := Q, P_2 := Q)$$

Suppose $\eta \mapsto *$ is in the denotation of the left hand side. Then there is a decomposition $\eta = \eta_r \cdot \eta_0 \cdot \eta_l$ such that $\eta_r \mapsto i \in \llbracket Q \rrbracket$ (for some $i \in |\delta|$), $\eta_0 \mapsto t \in \llbracket P_0 \rrbracket$ (for some $t \in \{tt, ff\}$), and $\eta_l \mapsto \text{put}.i \in \llbracket P_k \rrbracket$ (for $k = 1, 2$ depending on t). Now, η_r and η_0 are passive traces (because i and t are passive tokens). So, $\eta = \eta_0 \cdot \eta_r \cdot \eta_l$. Clearly, $\eta \mapsto *$ is in the denotation of the right hand side. Moreover, the argument is invertible. Hence, the two sides have the same denotation. ■

For the opposite direction, we define a notion of computable phrases:

Definition. A phrase P is said to be *computable* if one of the following conditions holds:

- Suppose P is a semi-closed phrase of type θ . If θ is a base type, P is computable if there exists P' such that $P \longrightarrow^* P'$ and $\llbracket P' \rrbracket$ is adequate for execution (in the sense of Lemma 1, forward direction). If θ is $\mathbf{var}[\delta]$, P is computable if $\mathbf{deref} P$ is computable and $P := E$ is computable for all semi-closed computable expressions E . If θ is $\theta_1 \times \theta_2$, P is computable if $\text{fst}(P)$ and $\text{snd}(P)$ are computable. If θ is $\theta_1 \rightarrow \theta_2$, P is computable if, for all semi-closed computable phrases Q , PQ is computable.
- A general phrase $x_1 : \theta_1, \dots, x_n : \theta_n \vdash P : \theta$ is computable if, for all semi-closed computable phrases Q_1, \dots, Q_n of appropriate types, $[Q_1, \dots, Q_n/x_1, \dots, x_n]P$ is computable.

LEMMA 3 *If $P \longrightarrow P'$ and P' is computable then P is computable.*

Proof: By induction on the type of P . ■

LEMMA 4 *If $\llbracket P \rrbracket = \emptyset$ then P is computable.*

Proof: By induction on the type of P . ■

LEMMA 5 *All phrases $\Pi \mid \Gamma \vdash P : \theta$ are computable.*

Proof: By induction on the type derivation of P . We need to show that, for all semi-closed computable substitutions σ that are correct for $\Pi \mid \Gamma$, $\sigma(P)$ is computable. The proof is a standard argument. (See, e.g., [21].) We show one case, $P = \mathbf{if}_\theta$, where a subsidiary induction on θ is involved. The goal is to show that for all semi-closed computable phrases P_0, P_1 and P_2 , the phrase $\mathbf{if}_\theta(P_0, P_1, P_2)$ is computable.

If θ is a base type, say \mathbf{comm} , then $\mathbf{if}_\theta(P_0, P_1, P_2) \longrightarrow^* \mathbf{if}_\theta(P'_0, P'_1, P'_2)$ whenever $P_i \longrightarrow^* P'_i$ (for $i = 0, 1, 2$). It is easy to see that if $\llbracket P'_i \rrbracket$ are adequate for execution then $\llbracket \mathbf{if}_\theta(P_0, P_1, P_2) \rrbracket$ is adequate for execution.

If θ is $\mathbf{var}[\delta]$, we must show that $\mathbf{deref} \mathbf{if}_\theta(P_0, P_1, P_2)$ is computable and that $\mathbf{if}_\theta(P_0, P_1, P_2) := E$ is computable for all semi-closed computable expressions E . For the first, use an argument similar to $\theta = \mathbf{comm}$ above. The second reduces in one step to $\mathbf{if}_{\mathbf{comm}}(P_0, P_1 := E, P_2 := E)$ whose computability is covered by the case of $\theta = \mathbf{comm}$ above. We have the conclusion by Lemma 3.

If $\theta = \theta_1 \rightarrow \theta_2$, we must show that, for all semi-closed computable phrases Q of type θ_1 , $\mathbf{if}_\theta(P_0, P_1, P_2) Q$ is computable. This term reduces in one step to $\mathbf{if}_{\theta_2}(P_0, P_1 Q, P_2 Q)$, which is computable by the inductive hypothesis for θ_2 . We have the conclusion by Lemma 3. ■

Proof of Proposition 9: Suppose C terminates, i.e., there exists C' such that $C \longrightarrow^* C'$ and $([], C') \Downarrow []$. It is easy to see (by induction on the structure

of C') that $([], \omega(C')) \Downarrow []$. Lemma 2 gives $\llbracket C \rrbracket = \llbracket C' \rrbracket$, and Lemma 1 shows $([] \mapsto *) \in \llbracket \omega(C') \rrbracket$. Since $\llbracket \omega(C') \rrbracket \subseteq \llbracket C' \rrbracket$, we have $([] \mapsto *) \in \llbracket C' \rrbracket$.

Conversely, if $([] \mapsto *) \in \llbracket C \rrbracket$, by Lemma 5, there exists a phrase C' such that $C \xrightarrow{*} C'$ and $\llbracket C' \rrbracket$ is adequate for execution. Since $[]$ is the only state for the empty type context and $[] \xrightarrow{\uparrow} []$, we have that $([], C') \Downarrow []$. Hence, C terminates. \blacksquare

Notes

1. We use the terminology of Algol 60 [41]: “variable” means a storage variable and “identifier” means a variable in the sense of lambda calculus; “expression” means a state-reading computation and “phrase” means a term in the sense of lambda calculus.
2. We use the term “object-based” rather than “object-oriented” to emphasize that our objects have state. The latter term is often used to describe languages with object inheritance, either with or without state.
3. The application of object-based semantics to full Algol is explored in the later work [47]. However, our experience suggests that this necessarily involves global states and functor categories.
4. The author is informed that P. W. O’Hearn developed this type system in 1991, but it was not made public because no semantic justification for the rule *Passify* was known. Its revival was made possible by the present semantics — more precisely a variant of it presented in [61], which was itself motivated by the subject reduction issues with SCI. This mutual reinforcement is an excellent example of the interplay between syntax and semantics.
5. This corresponds to the “copy rule” semantics for procedures described in the Algol 60 report [41]. There is, however, no notion of a standard reduction, and the normal form of a phrase is typically an “infinite phrase.” In practice, the reduction and execution phases of the semantics operate concurrently, with the execution phase demanding reduction steps when needed.
6. In an earlier version of this paper [58], two kinds of values called “passive” and “active” values were mentioned. “Objects” of the present paper subsume both these values, though our focus is mainly on active values.
7. This notion is consistent with the terminology used in the object-oriented programming community, e.g., [82]. On the other hand, note that we propose to *formalize* this notion in a way that has not been done before.
8. The notation “val.0” means the value 0, tagged with a symbol “val” for identification purposes. Mathematically, it is just an ordered pair (val, 0).
9. We are using the term “simulation” in an informal sense. In particular, the technical notion of “simulation” as in [38] is quite different. This is also the case for the recent use of “simulations” as morphisms in [13], [22], though there is a strong resemblance.
10. The elements of $A \otimes B$ do not correspond to any kind of pairs (which justifies the name “tensor” product). We use this construction in the context of linear functions (Section 4.2), where it will be seen that the elements are not of much consequence.
11. The stable order is often looked upon with suspicion because it gives inappropriate results for functional programming languages like PCF. However, we find that the stable order matches quite well with imperative programming languages. See Sec. 6.
12. In a previous version of this paper [58], the ! and † constructions on coherent spaces were used to model “passive values” and “active values” respectively. The move to active-passive spaces allows us to model both kinds of values under one heading. This gives a more general treatment of interference-controlled Algol than previously possible.

13. While it has been possible, in principle, to calculate the structure of such simple domains by syntactic means, there is no evidence that anybody has done so. The first instance where simple domains are calculated is in [50], where the calculation is done using semantic tools.
14. This observation means that Milner's context lemma fails for Algol (as well as most other programming languages with mutable variables), a known fact of folklore. From a categorical standpoint, this means that the fully abstract model of Algol is not a "well-pointed" category [40]. The functor category semantics of Reynolds and Oles [51], [66] seems to be the first work to reflect this fact.
15. Note that the full abstraction result claimed in [73] is for a language with such a snap-back combinator.
16. The soundness of such proof rules is not in question as other models already exist for specification logic [49], [78]. The point of interest is that such reasoning principles are valid in *our* model, while they would be invalid in models that support snap-back expressions.

References

1. I. J. Aalbersberg and G. Rozenberg. Theory of traces. *Theoretical Comput. Sci.*, 60:1–82, 1988.
2. S. Abramsky. Interaction categories and communicating sequential processes. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honor of C. A. R. Hoare*, pages 1–16. Prentice-Hall International, 1994.
3. S. Abramsky and R. Jagadeesan. New foundations for geometry of interaction. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 211–222. IEEE Computer Society Press, June 1992.
4. S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *J. Symbolic Logic*, 59(2):543–574, 1994.
5. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF (extended abstract). In *Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 1–15. Springer-Verlag, 1994.
6. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
7. K. R. Apt. Ten years of Hoare's logic: A survey. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, October 1981.
8. H. P. Barendregt. *The Lambda Calculus - Its Syntax and Semantics, 2nd Edition*. North-Holland, 1984.
9. P. N. Benton, G. M. Bierman, V. C. V. de Paiva, and J. M. E. Hyland. Term assignment for intuitionistic linear logic. Technical Report 262, Computer Laboratory, University of Cambridge, August 1992.
10. G. Berry. Stable models of typed λ -calculi. In *Fifth Intern. Colloq. Aut., Lang. and Program.*, volume 62 of *LNCS*, pages 72–88. Springer-Verlag, 1978.
11. G. M. Bierman. *On Intuitionistic Linear Logic*. PhD thesis, Computer Laboratory, University of Cambridge, December 1993.
12. S. Brookes, M. Main, A. Melton, and M. Mislove, editors. *Mathematical Foundations of Programming Semantics: Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theor. Comput. Sci.* Elsevier, 1995.
13. C. Brown and D. Gurr. A categorical linear framework for petri nets. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 208–218. IEEE Computer Society Press, June 1990.
14. R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Inf. Comput.*, 111(2):297–401, June 1994.
15. J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, volume 354 of *LNCS*. Springer-Verlag, 1989.

16. D. P. Freidman and D. S. Wise. An approach to fair applicative multiprogramming. In G. Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *LNCS*, pages 203–226. Springer-Verlag, 1979.
17. J.-Y. Girard. Linear logic. *Theoretical Comput. Sci.*, 50:1–102, 1987.
18. J.-Y. Girard. Towards a geometry of interaction. In Gray and Scedrov [20], pages 69–108.
19. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Univ. Press, 1989.
20. J. W. Gray and A. Scedrov, editors. *Categories in Computer Science and Logic*, volume 92 of *Contemp. Math.* AMS, 1989.
21. C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
22. V. Gupta. *Chu Spaces: A Model of Concurrency*. PhD thesis, Stanford University, August 1994.
23. J. Y. Halpern, A. R. Meyer, and B. A. Trahktenbrot. 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.
24. M. Hennessy. A fully abstract denotational model for higher-order processes. *InfComp*, 112(1):55–95, July 1994.
25. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–583, 1969.
26. C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symp. Semantics of Algorithmic Languages*, volume 188 of *Lect. Notes Math.*, pages 102–116. Springer-Verlag, 1971.
27. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
28. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, London, 1985.
29. J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF (preliminary version). Manuscript, Cambridge University, October 1994.
30. T. Jim and A. R. Meyer. Full abstraction and the context lemma. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 131–151. Springer-Verlag, 1991.
31. G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998, 1977.
32. R. M. Keller and G. Lindstrom. Approaching distributed database implementations through functional programming concepts. In *Intl. Conf. on Distributed Computing Systems*. IEEE, May 1985.
33. J. Lambek. Multicategories revisited. In Gray and Scedrov [20].
34. S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
35. A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.
36. A. Mazurkiewicz. Basic notions of trace theory. In de Bakker et al. [15], pages 285–363.
37. 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.
38. R. Milner. An algebraic definition of simulation between programs. In *Proc. Second Intern. Joint Conf. on Artificial Intelligence*, pages 481–489, London, 1971. The British Computer Society.
39. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
40. J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 365–458. North-Holland, Amsterdam, 1990.
41. P. Naur (ed). Report on the algorithmic language ALGOL 60. *Comm. ACM*, 3(5):299–314, May 1960.
42. 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.

43. P. W. O'Hearn. *Semantics of Noninterference: A Natural Approach*. PhD thesis, Queen's University, Kingston, Canada, 1990.
44. P. W. O'Hearn. Linear logic and interference control. In *Category Theory and Computer Science*, volume 350 of *LNCS*, pages 74–93. Springer-Verlag, 1991.
45. P. W. O'Hearn. A model for syntactic control of interference. *Math. Struct. Comput. Sci.*, 3:435–465, 1993.
46. P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In Brookes et al. [12].
47. P. W. O'Hearn and U. S. Reddy. Objects, interference and Yoneda embedding. In Brookes et al. [12].
48. 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 Univ. Press, 1992.
49. P. W. O'Hearn and R. D. Tennent. Semantical analysis of specification logic, Part 2. *Inf. Comput.*, 107(1):25–57, 1993.
50. P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3), 1995.
51. 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, 1985.
52. S. L. Peyton Jones and J. Launchbury. State in Haskell. *J. Lisp and Symbolic Comput.*, 1996. (to appear).
53. S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 1993.
54. B. C. Pierce and D. N. Turner. Object-oriented programming without recursive types. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 299–312. ACM, 1993.
55. G. Plotkin and G. Winskel. Bistructures, bidomains and linear logic. In *ICALP '94*, volume 820 of *LNCS*. Springer-Verlag, 1994.
56. G. D. Plotkin. LCF considered as a programming language. *Theoretical Comput. Sci.*, 5:223–255, 1977.
57. V. Pratt. The pomset model of parallel processes: Unifying the temporal and the spatial. In *Seminar on Concurrency*, volume 197 of *LNCS*, pages 180–196. Springer-Verlag, 1984.
58. U. S. Reddy. Global state considered unnecessary: Semantics of interference-free imperative programming. In *ACM SIGPLAN Workshop on State in Program. Lang.*, pages 120–135. Technical Report YALEU/DCS/RR-968, June 1993.
59. U. S. Reddy. A linear logic model of state. Electronic manuscript, University of Illinois (anonymous FTP from cs.uiuc.edu), October 1993.
60. U. S. Reddy. A linear logic model of state (extended abstract). Technical report, University of Glasgow, January 1993.
61. U. S. Reddy. Passivity and independence. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 342–352. IEEE Computer Society Press, July 1994.
62. U. S. Reddy and S. N. Kamin. Two semantic models of object-oriented languages. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 463–496. MIT Press, 1994.
63. C. Retoré. *Réseaux et Séquents Ordonnés*. Thèse de Doctorat, spécialité Mathématiques, Université Paris 7, February 1993. (English translation available from the author).
64. J. C. Reynolds. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.*, pages 39–46. ACM, 1978.
65. J. C. Reynolds. *The Craft of Programming*. Prentice-Hall International, London, 1981.
66. 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.
67. J. C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge Univ. Press, 1982.
68. J. C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie-Mellon University, June 1988.

69. J. C. Reynolds. Syntactic control of interference, Part II. In *Intern. Colloq. Aut., Lang. and Program.*, volume 372 of *LNCS*, pages 704–722. Springer-Verlag, 1989.
70. D. S. Scott. Relating theories of the lambda calculus. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 403–450. Academic Press, 1980.
71. E. Shapiro. *Concurrent Prolog: Collected Papers*. MIT Press, 1987. (Two volumes).
72. K. Sieber. New steps towards full abstraction for local variables. In *ACM SIGPLAN Workshop on State in Program. Lang.*, pages 88–100. Technical Report YALEU/DCS/RR-968, Yale University, New Haven, June 1993.
73. K. Sieber. Full abstraction for the second order subset of an Algol-like language (preliminary report). Technical Bericht A 01/94, Universitaet des Saarlandes, Saarbruecken, February 1994.
74. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
75. 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 *LNCS*, pages 192–214. Springer-Verlag, 1991.
76. R. D. Tennent. Semantics of interference control. *Theoretical Comput. Sci.*, 27:297–310, 1983.
77. R. D. Tennent. Denotational semantics of Algol-like languages. Internal Tech. Report 88-IR-02, Queen's University, September 1988. (To appear in Abramsky, S., Gabbay, D. M. and Maibaum, T. S. E. (eds) *Handbook of Logic in Computer Science*, Vol II, Oxford University Press).
78. R. D. Tennent. Semantical analysis of specification logic. *Inf. Comput.*, 85(2):135–162, 1990.
79. R. D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, London, 1991.
80. 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).
81. S. Weeks and M. Felleisen. On the orthogonality of assignments and procedures in Algol. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 1993.
82. P. Wegner. Dimensions of object-based language design. In *Object Oriented Prog. Syst., Lang. and Applications*. ACM SIGPLAN, 1987.
83. G. Winskel. An introduction to event structures. In de Bakker et al. [15], pages 364–397.
84. G. Winskel. Stable bistructure models of PCF. In *Math. Foundations of Computer Science*, volume 841 of *LNCS*. Springer-Verlag, 1994.
85. G.-Q. Zhang. *Logic of Domains*. Birkhäuser, Boston, 1991.
86. G.-Q. Zhang. Some monoidal closed categories of stable domains and event structures. *Math. Struct. Comput. Sci.*, 3:259–276, 1993.