# Objects and Classes in Algol-like Languages[1]

Uday S. Reddy
*The University of Illinois at Urbana-Champaign*

Many object-oriented languages used in practice descend from Algol. With this motivation, we study the theoretical issues underlying such languages via the theory of Algol-like languages. It is shown that the basic framework of this theory extends cleanly and elegantly to the concepts of objects and classes. Moreover, a clear correspondence emerges between classes and abstract data types, whose theory corresponds to that of existential types. Equational and Hoare-like reasoning methods, and relational parametricity provide powerful formal tools for reasoning about Algol-like object-oriented programs.

*Key Words:* Algol-like languages, relational parametricity, specification logic, object-oriented programming, semantics.

## 1. INTRODUCTION

Object-oriented programming first developed in the context of Algol-like languages in the form of Simula 67 [17]. The majority of object-oriented languages used in practice either descend from Algol or use ideas from the Algol tradition. Thus, it seems entirely appropriate to study the concepts of object-oriented programming in the context of Algol-like languages. This paper is an effort to formalize how objects and classes are used in Algol-like languages and to develop their theoretical underpinnings.

Our formal framework is based on Reynolds's analysis of "Algol-like languages." The *Idealized Algol* of Reynolds is a typed lambda calculus with base types that support state-manipulation (for expressions, commands, etc.). The typed lambda calculus framework gives a "mathematical" flavor to Idealized Algol and sets it within the broader programming language research. Yet, the base types for state-manipulation make it remarkably close to popular programming languages. This combination gives us an ideal setting for studying various programming language phenomena of relevance to languages like C++, Modula-3 and Java etc.

Reynolds also argued [59, Appendix] that object-oriented programming concepts are implicit in his Idealized Algol. The essential idea is that classes correspond

[1]This research was supported by National Science Foundation grant CCR-96-33737.

to "new" operators that generate instances every time they are invoked. This obviates the need for a separate "class" concept. The idea has been echoed by others [56, 2]. In contrast, we take here the position that there is significant benefit to directly representing object-oriented concepts in the formal system instead of encoding them by other constructs. While the *effect* of classes can be obtained by their corresponding "new" operators, not all *properties* of classes are exhibited by the "new" operators. Thus, classes form a specialized form of "new" operators that are of independent interest.

In this paper, we define a language called $IA^+$ as an extension of Idealized Algol for object-oriented programming and study its semantics and formal properties. An important idea that emerges, from the view point of Algol theory, is that classes are *abstract data types* whose theory corresponds to that of existential types as in SOL [43]. (While the intuitive connection between classes and abstract types is well-known and dates back to Hoare's early insights [28], a formal theory of classes comparable to that of SOL has not been previously available.) In a sense, $IA^+$ is to Idealized Algol what SOL is to polymorphic lambda calculus. Like SOL, it adds types and features that explicitly represent data abstraction. However, while SOL can be faithfully encoded in polymorphic lambda calculus [55], the data abstraction features of $IA^+$ are more refined than those expressible in Idealized Algol. The corresponding encoding does not preserve equivalences. Thus, $IA^+$ is a proper extension.

*Related work*

In the earlier work of the author [56, 32], a global state-based semantics was defined for stateful object-oriented programs. Being a global state-based semantics, it does not handle the state encapsulation issues of objects adequately. The deficiencies of the global state set-up have been discussed in a number of papers [39, 51, 58]. Work on specification of stateful objects includes [6, 34, 35, 36] in addressing subtyping issues and [3, 7] in addressing self-reference issues.

A number of recent papers [1, 8, 11, 19, 18] discuss object-oriented type systems for languages with side effects, but this work does not address reasoning principles for programs. A related direction is that of "object encodings" which might be thought of as syntactic presentations of semantics. Pierce and Turner [54] study the encoding of objects as abstract types, which bears some similarity to the parametricity semantics in this paper. More recent work along these lines is [12]. Fisher and Mitchell [21, 20] also relate classes to data abstraction, though this seems to be at a different level than that discussed here. All this work is usually carried out in a functional setting for objects, but some of the ideas deal with "state."

The major developments in the research on Algol-like languages are collected in [52]. Tennent [67] gives a gentle introduction to the concepts as of 1994.

## 2. OBJECTS

Object-oriented programming involves several novel concepts that are of interest from a semantic point of view. The foremost among them is the notion of *state encapsulation*. This is the idea that objects encapsulate some physical resources, typically memory locations, and provide operations to manipulate these resources. State encapsulation gives rise to *data abstraction* because the encapsulated re-

sources are not accessible to client programs except via the exported operations. This form of data abstraction was first studied by Hoare [28], but it was formalized for full Algol-like languages only recently by O'Hearn and Tennent [51] using the theory of relational parametricity. Explicating this theory for object-oriented languages with classes is the main focus of this paper.

A second novel concept of object-oriented programming is the notion of *self-reference* and how it interacts with inheritance. One of the well-understood semantic models for these concepts is in terms of recursion and fixed points, studied by Cardelli [13], Cook [15] and Reddy [56]. The recursion model can be readily adapted to Algol-like languages because it works within a typed lambda calculus framework. We will point out how this goes. (Another semantic model for self-reference is in terms of self-application [31, 32] which has received much attention in the Abadi-Cardelli calculus of objects [2]. We do not consider the self-application model in this paper.)

A third important concept in object-oriented programming is the notion that objects form *dynamic data*. All objects have *references* that uniquely identify them, and these references can be assigned to variables and manipulated dynamically. While dynamic data structures are pervasive in traditional languages of the Algol family (e.g., in Algol W, Simula, Pascal and Ada), their theoretical foundations are only now beginning to be studied [23], and much work remains to be done. So, we omit the treatment of references from the main body of the paper, except to note how they can be incorporated in an Algol-like type system. The issues of state encapsulation are, however, present in all object-oriented languages used in practice.

In this section, we describe these issues informally in order to motivate the formal treatment that follows in the remaining sections.

An *object* is a programming abstraction that encapsulates some physical resources — such as memory locations, input/output streams and other devices — and exports operations to manipulate these resources. The exported operations are called the "methods" of the object. Anticipating type systems that allow us to group all the methods together into a unit, we call such a group a "method suite." The resources encapsulated by an object are said to comprise its "state." In a language with dynamic data, an object would also have a "reference" which is assigned when the object is created and uniquely identifies the object.

Two attributes of an object are of semantic interest:

- its *type*, which describes the interface of the object as manipulated through its methods, and
- its *class*, which determines the behavior of the object.

As an example, consider a *counter* object that remembers an integer count and provides operations for reading and incrementing the value of the count. We might define its type as

$$\textbf{type}\ \mathit{counter} = \{\mathit{val} : \mathsf{exp}[\mathsf{int}], \mathit{inc} : \mathsf{comm}\}$$

The *val* method, which reads the count, is an "expression" in Algol terminology. It *reads* the state of the counter to produce an integer. The *inc* method is a

"command" which *transforms* the state of the counter. The two methods are grouped together into a record, signified by braces $\{\ldots\}$.

There is nothing in the type of counters that describes their behavior (i.e., what the *val* method reads and what the *inc* method does). The specification of such behavior constitutes the *class* of the counter. Note that the state encapsulated by a counter object is invisible to the client programs. Thus, differences in the encapsulated state should be factored out in specifying the class. We consider two kinds of class descriptions: a *state-based* description, where the behavior is specified in terms of a hypothetical state set, and an *event-based* description, where the behavior is specified in terms of events observed via method invocation. Both of these descriptions are *semantic* concepts. Syntactic notations for describing classes will be discussed in the sequel.

*State-based descriptions of classes*

A *state machine* for counter objects can be described by giving

- a state set $Q$,
- the initial state when the counter is created, $q_0 \in Q$, and
- the effect of the methods on the counter state.

For our chosen methods, the effect of *val* is given by a function of type $Q \to \mathit{Int}$ and the effect of *inc* is given by a function of type $Q \to Q$. (We are ignoring the issues of divergence and recursion.) For example, a state machine for counters can be:

$$M = \langle \mathit{Int}, 0, \{\mathit{val} = \lambda n.\, n,\ \mathit{inc} = \lambda n.\, n+1\}\rangle$$

Here the state set is the set of integers, 0 is the initial state, the *val* method returns the integer state and the *inc* method increments the integer state. Another state machine for counters is:

$$M' = \langle \mathit{Int}, 0, \{\mathit{val} = \lambda n.\, (-n),\ \mathit{inc} = \lambda n.\, n-1\}\rangle$$

The difference from $M$ is that the *inc* method *decrements* the integer state (so that successive increments trace through the sequence $0, -1, -2, \ldots$). However, the *val* method negates the integer state to give its output. So, the overall behavior described by $M'$ is the same as that described by $M$. We say that $M$ and $M'$ are *behaviorally equivalent*.

The equivalence of $M$ and $M'$ can be established by exhibiting a *simulation relation* $R$ between the two state sets:

$$n\, R\, n' \iff n \geq 0 \wedge n' = -n \tag{1}$$

The relation $R$ relates the states in the two machines that have equivalent observable effect. We see that the two *val* operations give equal results for $R$-related states and the two *inc* operations map $R$-related states to $R$-related states. This is stated more formally as

$$M.\mathit{val}\ [R \to \Delta_{\mathit{Int}}]\ M'.\mathit{val}$$
$$M.\mathit{inc}\ [R \to R]\ M'.\mathit{inc}$$

where $\Delta_{Int}$ is the equality relation for *Int* and the relational operator $\rightarrow$ says that related inputs are mapped to related outputs. The machines $M$ and $M'$ are said to be *similar* (by virtue of the simulation relation). Behavioral equivalence is the transitive closure of similarity.

A state-based description of a class consists of an equivalence class of state machines under behavioral equivalence. By giving a state machine, such as $M$ or $M'$, we uniquely describe its equivalence class. The state set used in the description is "hypothetical" in the sense that it does not form an essential part of the behavior but is used as a tool in describing the behavior. Different state sets can be used in different ways to describe the same behavior.

*Event-based description of classes*

Since the state sets are incidental in describing object behavior, it is natural to ask if a state-free description of the behavior can be given. Indeed, in automata theory, the behavior of a state machine can be described in terms of the language accepted by the machine or the sequential function computed by the machine without reference to states. This approach has also been used in concurrency theory to good effect [29, 41]. A similar approach can be used for objects but, since the operations of objects are of complex types, the vocabulary used for their description is more sophisticated. The basic structure of such vocabulary originates from Winskel's event structures [68] though the recently developed game semantics can be used to give more refined descriptions [5].

For the expository treatment of this section, we indicate how events can be used to describe object behavior, leaving further details to Section 5.2. An "event" represents the information exchanged between an object and a client program during a method invocation. Different types have different events associated with them (because the information exchanged depends on the type). Moreover, events for compound types are built from events for their constituent types.

For example, events for the type $\mathsf{exp}[\delta]$ are just $\delta$-typed data values. Events for comm include a single event '$*$' denoting the successful completion of a command execution. Even though the execution of a command transforms the state, no information about the transformation is directly exchanged by the object and the client. Thus, the only event directly observable by running a command is its termination. Events for a record type $\{m_1 : \theta_1, \ldots, m_n : \theta_n\}$ are pairs $(m_i, d)$ where $m_i$ is a field name and $d$ is an event appropriate for the corresponding type $\theta_i$. The event $(m_i, d)$ denotes the action of a client program selecting the $m_i$ field and then constructing an event $d$ in the process of using this field.

We refer to a sequence of events of a particular type as an *event trace.* The set of event traces observable from an object is called its *trace set.* The trace set constitutes a state-free description of the object behavior. For example, the trace set of a counter object is shown in Figure 1 in diagrammatic form. (The traces in the set are the sequences of labels of all paths starting from the top node.). The events for this object are "(inc, $*$)" denoting a successful completion of the *inc* method, and "(val, $i$)" denoting a completion of the *val* method with the result $i$ (an integer). The nodes can be thought of as (abstract) states and events as state transitions. Note that a *val* event does not change the state whereas an *inc* event takes the object to a state with a higher *val* value. For discussion purposes, we

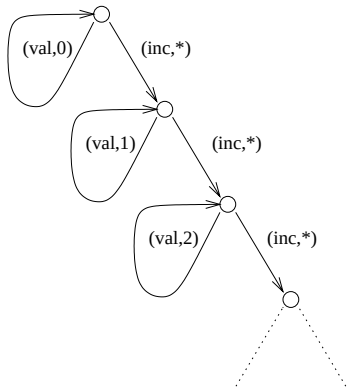


**FIG. 1.** Trace set of a counter object

can label each node with an integer (which might well be the same integer given by *val*). The trace set can then be described mathematically by a recursive definition:

$$\begin{aligned} &\textsc{cnt}(0) \text{ where} \\ &\quad \textsc{cnt}(n) = \{\epsilon\} \cup \{(\text{inc}, *)\} \cdot \textsc{cnt}(n+1) \\ &\qquad\qquad \cup \{(\text{val}, n)\} \cdot \textsc{cnt}(n) \end{aligned}$$

The parameter of the CNT function is the label of the state. Note that these labels can be anything we make up, but often it makes sense to use labels that correspond to states in an implementation. For instance, here is another description of the same trace set using negative integers for labels:

$$\begin{aligned} &\textsc{cnt}'(0) \text{ where} \\ &\quad \textsc{cnt}'(n) = \{\epsilon\} \cup \{(\text{inc}, *)\} \cdot \textsc{cnt}'(n-1) \\ &\qquad\qquad \cup \{(\text{val}, (-n))\} \cdot \textsc{cnt}'(n) \end{aligned}$$

This description corresponds to the state machine $M'$. While it is obvious that the two trace sets are the same, a formal proof would use the simulation relation $S$ defined in (1). We can show by fixed point induction that

$$n \; S \; n' \implies \textsc{cnt}(n) = \textsc{cnt}'(n')$$

and it follows that $\textsc{cnt}(0) = \textsc{cnt}'(0)$.

Note that in this description there is virtually no difference between classes and instances. A class determines a trace set which is then shared by all instances of the class.

The two forms of class descriptions play complementary roles. While the state-machine description gives a closer connection to implementations by focusing on the internal structure of objects, the event-based description gives a more abstract view in terms of the observable behavior. The latter would be more appropriate, for instance, in a distributed setting where objects might have complex internal structure but simple interfaces.

Object behavior, specified either in terms of state machines or trace sets, constitutes a class. Any object with the specified behavior is said to be an *instance* of this

class. Note that all instances of a class have exactly the same behavior. However, each of them encapsulates separate physical resources to maintain its state. Hence, each has its own path of evolution independent of all other instances of the class. This is the *only* difference between different instances of a class.

*Class implementations*

The two methods of class description mentioned above are meant for building abstract conceptual models of classes. Within the programming language, classes are defined by giving *implementations*. We implement objects of a new class by using one or more local objects of previously defined classes, and writing a term for the method suite which invokes the methods of these local objects. The objects used in building the new object are *local* to the new object in that they are inaccessible to the client of the new object except via the methods. For example, a class implementation for counters might be of the form:

```
Counter = class:
                {inc: comm, val: exp[int]}
          local
                Var[int] cnt
          init
                cnt := 0
          meth
                {inc = (cnt := cnt + 1),
                 val = cnt }
```

Counter objects are implemented here using an integer variable as a local object. The *inc* and *val* methods are defined by appropriate terms of type comm and exp[int] respectively. The **init** term serves to initialize the state of the local object. It is not hard to see that any such class implementation determines an abstract state machine which in turn determines a class behavior.

*Types versus classes*

In most object-oriented languages of the Algol family, classes are regarded as types. On the other hand, our analysis brings out types and classes as distinct concepts. So, this divergence warrants some comment.

One reason for treating classes as types is that it gives tight control over which objects are regarded as belonging to a type. This is not the case with interface types. For example, even though we used the name *counter* as an abbreviation for the type {inc: comm, val: exp[int]}, an arbitrary record of this type need not behave anything like a counter. On the other hand, all instances of the class *Counter* have the behavior of counters. Thus, by treating the class *Counter* as a type, we obtain tighter control over values of the type.

However, the class *Counter* is a particular implementation of the abstract behavior of counters. We can define another class, e.g., one that corresponds to the state machine $M'$, which has the same behavior as *Counter*. In a type system that regards classes as types, the two classes would be regarded as distinct types even though they describe the same behavior. Since one would like to be able to freely

interchange different implementations of the same behavior, this would seem to be too limiting.

An appropriate solution that combines the advantages of both the approaches is to use abstract types. By postulating *counter* as an abstract interface type that is implemented by the class *Counter*, we retain the flexibility of defining other classes that implement the same interface. Since this solution is entirely consistent with our approach of treating interfaces as types, we continue to use interface types in the main body of the paper. In section 6, we discuss how to add abstract interface types to the type system.

Many of the types and type constructors typically found in Algol-like languages, such as variables, arrays and records, appear as classes and class constructors in our formulation. The reason is that these so-called "types" determine not only the interface but also the behavior of the corresponding data objects. Typical "declarations" in these languages are instance-creation operations, not type declarations. It may be seen that our analysis sheds light on the nature of "types" and "declarations" in these languages.

## 3. THE LANGUAGE IA$^+$

The language IA$^+$ is an extension of Idealized Algol with classes. Thus, it is a typed lambda calculus with base types corresponding to imperative programming phrases. The base types include:

- comm, the type of *commands* or state-transformers, and
- exp$[\delta]$, the type of state-dependent *expressions* giving $\delta$-typed values,
- val$[\delta]$, the type of phrases that directly denote $\delta$-typed *values* (without any state-dependence).

Here, $\delta$ ranges over a collection of *data types* such as int(eger) and bool(ean) whose values are storable in variables. The "types" like exp$[\delta]$ and comm are called "phrase types" to distinguish them from data types. Values of arbitrary phrase types are not storable in variables.[2]

An important principle of Algol-like languages is that the types of terms precisely demarcate the effects that terms might have. For example, the only terms that transform the state are those of type comm. Terms that can read the state are those of types comm or exp$[\delta]$. On the other hand, terms of type val$[\delta]$ and those of other phrase types like function types do not read or write the state.

The collection of *phrase types* (or "types," for short) is given by the following syntax:

$$\theta \quad ::= \quad \beta \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2 \mid \{x_1{:}\,\theta_1, \ldots, x_n{:}\,\theta_n\} \mid \mathsf{cls}\ \theta$$

where $\beta$ ranges over base types (exp$[\delta]$, comm and val$[\delta]$). Except for cls $\theta$ types, the remaining type structure is that of simply typed lambda calculus with record types

[2] It is possible to postulate a data type of references (or pointers) ref $\theta$, for every phrase type $\theta$, whose values are storable in variables. This obtains the essential expressiveness that the object-oriented programmer desires. Unfortunately, our theoretical understanding of references is not well-developed. So, we omit them from the main presentation and mention issues relating to them in Sec. 6.2.

$$\frac{}{\theta <: \theta} \qquad \frac{\theta <: \theta' \quad \theta' <: \theta''}{\theta <: \theta''}$$

$$\frac{\theta_1 <: \theta_1' \quad \theta_2 <: \theta_2'}{(\theta_1 \times \theta_2) <: (\theta_1' \times \theta_2')} \qquad \frac{\theta_1' <: \theta_1 \quad \theta_2 <: \theta_2'}{(\theta_1 \to \theta_2) <: (\theta_1' \to \theta_2')}$$

$$\frac{\theta <: \theta'}{\mathsf{cls}\ \theta <: \mathsf{cls}\ \theta'}$$

$$\frac{\theta_1 <: \theta_1' \quad \cdots \quad \theta_n <: \theta_n'}{\{x_1 : \theta_1, \ldots, x_n : \theta_n, \ldots, x_m : \theta_m\} <: \{x_1 : \theta_1', \ldots, x_n : \theta_n'\}}$$

$$\frac{}{\mathsf{val}[\delta] <: \mathsf{exp}[\delta]} \qquad \frac{}{\{\mathrm{get} : \mathsf{exp}[\delta],\ \mathrm{put} : \mathsf{val}[\delta] \to \mathsf{comm}\} <: \mathsf{exp}[\delta]}$$

**TABLE 1**
**Subtyping rules**

and subtyping. See, for instance, Mitchell [42, Ch. 10] for details. The type $\mathsf{cls}\ \theta$ is the type of classes that describe the behavior of $\theta$-typed objects. The subtyping rules of IA$^+$ are shown in Table 1. The basic subtypings are the following:

- $\mathsf{val}[\delta] <: \mathsf{exp}[\delta]$ regards a state-independent value as a state-dependent expression;
- $\mathsf{var}[\delta] <: \mathsf{exp}[\delta]$, where $\mathsf{var}[\delta] = \{\mathrm{get} : \mathsf{exp}[\delta],\ \mathrm{put} : \mathsf{val}[\delta] \to \mathsf{comm}\}$ denotes the signature type of variables, supports the implicit selection of the *get* operation; and
- record subtyping includes "width subtyping," whereby a longer record type is considered a subtype of a shorter record type, and "depth subtyping," whereby subtyping of fields propagates to the record types as a whole.

Our interpretation of subtyping is by *coercions* [42, Sec. 10.4.2]. For example, the width subtyping of records is interpreted by the forgetting-fields coercion.

The standard parameter passing mechanism of IA$^+$ is call by name (as is usual with typed lambda calculus). It is possible to incorporate Algol-style call by value via primitive operations.

*Classes*

For defining classes, we use a notation of the form:

**class**: $\theta$
  **local** $C_1\ x_1; \ldots; C_n\ x_n$
  **init** $A$
  **meth** $M$

The various components of the description are as follows:

- $\theta$ is a type (the type of all instances of this class), called the *signature* of the class,

- $x_1, \ldots, x_n$ are identifiers (for the local objects),
- $C_1, \ldots, C_n$ are terms denoting classes (of the respective local objects),
- $A$ is a comm-typed term (for initializing the local objects), and
- $M$ is a term of type $\theta$ (defining the methods of the class).

Admittedly, this is a complex term form but it represents quite closely the term forms for classes in typical programming languages. Moreover, we will see that much of this detail has a clear type-theoretic basis.

Any instance of a class thus defined contains $n$ local objects encapsulated within it (of classes $C_1, \ldots, C_n$ respectively), and exports a method suite denoted by $M$. The initialization command $A$ is used to initialize the local objects of the instance. Note that it would not be enough to just declare the types of the local objects (as opposed to their classes) because the types determine only the interface, not the behavior.

By default, the local objects declared in a class are "private," i.e., not part of the exported method suite. However, it is possible, if need be, to define a method that gives direct access to a local object.

It is noteworthy that we cannot define nontrivial classes without first having some primitive classes (needed for defining local objects). We will assume a primitive class of (mutable) variables for each data type $\delta$, via the constant:

$$\begin{aligned} &\mathrm{Var}[\delta] : \mathsf{cls}\ \mathsf{var}[\delta] \\ &\quad \text{where } \mathsf{var}[\delta] = \{\mathrm{get} : \mathsf{exp}[\delta],\ \mathrm{put} : \mathsf{val}[\delta] \to \mathsf{comm}\} \end{aligned}$$

If $x$ is an instance of $\mathrm{Var}[\delta]$ (a "variable"), then $x.\mathrm{get}$ is a state-dependent expression that gives the value stored in $x$ and $x.\mathrm{put}(k)$ is a command that stores the value $k$ in $x$.[3] The subtyping $\mathsf{var}[\delta] <: \mathsf{exp}[\delta]$ allows us to write simply $x$ where $x.\mathrm{get}$ is meant (often called implicit "dereferencing").

The *Counter* class mentioned in Section 2 gives an example of a defined class. It also illustrates the use of the the variable class. In writing

$$\mathrm{cnt} := \mathrm{cnt} + 1$$

we have used the subtyping $\mathsf{var}[\delta] <: \mathsf{exp}[\delta]$ for the occurrence of *cnt* on the right hand side. We could have written *cnt.get* to make this conversion explicit. The ":=" operator itself is a defined operation which invokes the put method of the variable (discussed below).

For creating instances of classes, we use the notation:

**new** $C$

which is a value of type $(\theta \to \mathsf{comm}) \to \mathsf{comm}$ where $\theta$ is the signature type of class $C$. For example,

**new** Counter $\lambda$a. $B$

[3] We assume that all new variables come initialized to some specific initial value $\mathsf{init}_\delta$. It is also possible to use a modified primitive $\mathrm{Var}[\delta]{:}\,\mathsf{val}[\delta] \to \mathsf{cls}\,\mathsf{var}[\delta]$ that allows explicit initialization via a parameter.

creates an instance of *Counter*, binds it to $a$ and executes the command $B$.[4] The partial phrase

**new** Counter λa.

is called an instance declaration. The effect of the declaration is roughly equivalent to the Java locution:

```
final Counter = new Counter();
```

However, there are no references (pointers) involved in our term. The identifier $a$ is directly bound to the *Counter* object whereas, in the Java version, $a$ is a variable that holds a reference to the newly created object.

*Remark.* The type of **new** $C$ illustrates how the "physical" nature of objects is reconciled with the "mathematical" character of Algol. If **new** $C$ were to be regarded as a value of type $\theta$ then the mathematical nature of Algol would prohibit stateful objects entirely. For example, a construction of the form

**let** a = **new** Counter
**in** a.inc; print a.val

would be useless because it would be equivalent, by $\beta$-reduction, to:

(**new** Counter).inc; print (**new** Counter).val

thereby implying that every use of $a$ gives a new counter and no state is propagated. The higher-order type of **new** $C$ gives rise to no such problems. This insight is due to Reynolds [60] and has been used in several other languages [45, 65].

One would want a variety of combinators for classes. The following polymorphic "product" combinator for making pairs of objects is an essential primitive:

$$* : \mathsf{cls}\ \theta_1 \times \mathsf{cls}\ \theta_2 \to \mathsf{cls}\ (\theta_1 \times \theta_2)$$

If $C_1$ and $C_2$ are classes then $C_1 * C_2$ is a class whose instances are pairs consisting of an instance of $C_1$ and an instance of $C_2$. So, the declaration

**new** $(C_1 * C_2)$ λ(x, y).

binds $x$ and $y$ to new instances of $C_1$ and $C_2$ respectively. The "$*$" combinator is intuitively similar to the product constructor of types, but it operates on classes. Since classes represent (the equivalence classes of) state machines, the product operation of classes is semantically quite different from products of types. (Cf. Section 5.1.)

Common data structures in programming languages such as arrays and records also give rise to class combinators. The constructor for arrays can be regarded as a combinator of type:

$$\mathrm{Array} : \mathsf{cls}\ \theta \to \mathsf{val}[\mathsf{int}] \to \mathsf{cls}\ (\mathsf{val}[\mathsf{int}] \to \theta)$$

[4] We use the convention that the scope of a lambda abstraction extends as far to the right as possible, often terminated by a closing parenthesis. We do not let ";" terminate the scope.

If $C$ is a class and $n$ an integer value, (*Array* $C$ $n$) is equivalent to the $n$-fold class product $C * \cdots * C$. Its instances are vectors of the form $(a_1, \ldots, a_n)$ where each $a_i$ is an independent instance of $C$. We regard such vectors as (partial) functions from integers to $C$-objects so that we can use the "subscripting" notation $a(i)$ to select the $i$' th component.

The Pascal-like record construction

**record** $C_1\ x_1; \ldots; C_n\ x_n$ **end**

is a variant of the class product $C_1 * \cdots * C_n$. If $C_1, \ldots, C_n$ are classes of types $\mathsf{cls}\ \theta_1$, ..., $\mathsf{cls}\ \theta_n$ respectively, then **record** $C_1\ x_1; \ldots; C_n\ x_n$ **end** is a class of type $\mathsf{cls}\ \{x_1{:}\,\theta_1, \ldots, x_n{:}\,\theta_n\}$. So, its instances are *records* with fields named $x_1, \ldots, x_n$. This compares with the class product $C_1 * \cdots * C_n$ whose instances are tuples of type $\theta_1 \times \cdots \times \theta_n$.

The recursion mechanism of the language provides for self-reference in class definitions. A class for describing self-referential objects of type $\theta$ is typically of type $\mathsf{cls}\ (\theta \to \theta)$, which allows the method suite to be parameterized by "self." For example, a class for counter objects with a "set" method may be defined as follows:

```
type setcounter = {set: val[int] → comm, inc: comm, val: exp[int]}
SetCounter =
  class:   setcounter → setcounter
    local Var[int] cnt
    init cnt := 0
    meth
        λself. {val = cnt.get,
                set = cnt.put,
                inc = self.set(self.val + 1)}
```

The method suite is parameterized by an object *self*, and the *inc* method invokes the methods of this object rather than reading and writing the local variable. The fixed point of the method-suite forms an object that has the desired behavior of counters.

We can define a generic combinator for taking such fixed-points:

```
close: cls (θ → θ) → cls θ
close c = class: θ local c f init skip meth (fix f)
```

The *close* combinator converts a self-referential class $C$ to an ordinary class whose instances invoke their own methods recursively. Now, a declaration of the form:

**new** (close SetCounter) λa.

binds $a$ to a counter object.

Another interesting application of the recursion mechanism is for creating inter-linked objects that invoke each other's methods. Such inter-linked objects arise in simulation applications as well as in graphical user interfaces. See, for example, [4, Sec. 3.3.4] and the Observer pattern in [22]. Consider an "inter-link" operator <>

defined as follows:

$$
\begin{array}{l}
\texttt{<>} : \mathsf{cls}(\theta_1 \times \theta_2 \to \theta_1) \times \mathsf{cls}(\theta_2 \times \theta_1 \to \theta_2) \to \mathsf{cls}\ (\theta_1 \times \theta_2 \to \theta_1 \times \theta_2) \\
C_1 \texttt{<>} C_2 = \textbf{class} :\ \theta_1 \times \theta_2 \to \theta_1 \times \theta_2 \\
\qquad\qquad \textbf{local}\ C_1\ f_1;\ C_2\ f_2 \\
\qquad\qquad \textbf{init}\ \mathsf{skip} \\
\qquad\qquad \textbf{meth}\ \lambda(x, y).\ (f_1(x, y),\ f_2(y, x))
\end{array}
$$

Here, $C_1$ and $C_2$ are classes whose method suites are parameterized by two objects: the first is the "self" object and the second is some other object that is meant to be inter-linked. Now, an instance of the class $close(C_1 \texttt{<>} C_2)$ is a pair of objects $x$ and $y$ which invoke each other's methods in a mutually recursive fashion.

Inheritance is accomplished by record-update with due attention paid to self reference. For record-update, we use the term form

$$M_1\ \textbf{with}[\tau]\ M_2$$

where $M_1$ is a record of type $\theta$ and $M_2$ is a record of type $\tau$, which denotes the record obtained by updating $M_1$ with $\tau$-fields from $M_2$. (Any extra fields in $M_2$, not mentioned in $\tau$, are ignored.) This is essentially the update operation of [16] but adjusted to treat record subtyping correctly. (Cf. [20] for a discussion of the last issue.)

As an example, considered a counter class that prints a warning when a preset limit is reached:

```
LimitCounter lim =
  class: setcounter → setcounter
    local SetCounter f
    init skip
    meth
      λself. (f self) with[set: val[int] → comm]
          {set = λk. if k ≤ lim then
                       (f self).set k
                     else print "Limit reached" }
```

This is defined as a derived class of the class *SetCounter* with an updated *set* method that forces the counter to stay within the limit. An instance of $close(LimitCounter)$ contains the updated *set* method. Moreover, since the *inc* method is defined in terms of *set*, any use of the *inc* method also respects the limit. This modeling of inheritance is due to Cook [15] and Reddy [56].

*Term syntax*

The type rules of IA$^+$ are shown in Table 2. The typed lambda calculus aspects of IA$^+$ are standard. As to cls types, we have one rule for introduction and one for elimination, whose term forms are class definition and instance declaration. We show a single local object in a **class** term for simplicity. This is obviously not a limitation because the $*$ combinator of classes can be used to create multiple local objects.

$$\frac{}{\Gamma, x{:}\theta \rhd x : \theta}\ \text{Id} \qquad \frac{}{\Gamma \rhd c : \theta}\ \text{Const} \qquad \frac{\Gamma \rhd M : \theta}{\Gamma \rhd M : \theta'}\ \text{Subs}\ \ (\text{if } \theta <: \theta')$$

$$\frac{\Gamma \rhd M : \theta \quad \Gamma \rhd N : \theta'}{\Gamma \rhd \langle M, N\rangle : \theta \times \theta'}\ \times\text{Intro} \qquad \frac{\Gamma \rhd M : \theta_1 \times \theta_2}{\Gamma \rhd \pi_i\, M : \theta_i}\ \times\text{Elim}\ (i = 1, 2)$$

$$\frac{\Gamma \rhd M_i : \theta_i \quad (i = 1, \ldots, n)}{\Gamma \rhd \{x_1 = M_1, \ldots, x_n = M_n\} : \{x_1{:}\theta_1, \ldots, x_n{:}\theta_n\}}\ \{\ \}\ \text{Intro}$$

$$\frac{\Gamma \rhd M : \{x_1{:}\theta_1, \ldots, x_n{:}\theta_n\}}{\Gamma \rhd M.x_i : \theta_i}\ \{\ \}\ \text{Elim}$$

$$\frac{\Gamma \rhd M : \{\vec{x}{:}\vec{\tau},\ \vec{y}{:}\vec{\theta}\} \quad \Gamma \rhd N : \{\vec{y}{:}\vec{\theta'},\ \vec{z}{:}\vec{\sigma}\}}{\Gamma \rhd M\ \mathbf{with}[\{\vec{y}{:}\vec{\theta'},\ \vec{z}{:}\vec{\sigma}\}]\ N : \{\vec{x}{:}\vec{\tau},\ \vec{y}{:}\vec{\theta'},\ \vec{z}{:}\vec{\sigma}\}}\ \{\ \}\ \text{Update}$$

(where $\vec{x}$ and $\vec{z}$ have no common identifiers)

$$\frac{\Gamma, x{:}\theta \rhd M : \theta'}{\Gamma \rhd \lambda x.\ M : (\theta \to \theta')}\ \to\text{Intro} \qquad \frac{\Gamma \rhd M : \theta \to \theta' \quad \Gamma \rhd N : \theta}{\Gamma \rhd M\, N : \theta'}\ \to\text{Elim}$$

$$\frac{\Gamma \rhd C : \mathsf{cls}\,\tau \quad \Gamma,\ x{:}\tau \rhd A : \mathsf{comm} \quad \Gamma,\ x{:}\tau \rhd M : \theta}{\Gamma \rhd (\mathbf{class}:\ \theta\ \mathbf{local}\, C\ x\ \mathbf{init}\ A\ \mathbf{meth}\, M) : \mathsf{cls}\,\theta}\ \mathsf{cls}\ \text{Intro}$$

$$\frac{\Gamma \rhd C : \mathsf{cls}\,\theta}{\Gamma \rhd \mathbf{new}\ C : (\theta \to \mathsf{comm}) \to \mathsf{comm}}\ \mathsf{cls}\ \text{Elim}$$

**TABLE 2**
**Type rules of IA$^+$**

| | | |
|---|---|---|
| $\mathsf{skip}$ | : | $\mathsf{comm}$ |
| $\_;\_$ | : | $\mathsf{comm} \times \mathsf{comm} \to \mathsf{comm}$ |
| $\mathbf{letval}_{\delta,\beta}$ | : | $\mathsf{exp}[\delta] \to (\mathsf{val}[\delta] \to \beta) \to \beta$ (where $\beta = \mathsf{exp}[\delta']$ or $\mathsf{comm}$) |
| $\mathbf{if}_\theta$ | : | $\mathsf{val}[\mathsf{bool}] \to \theta \to \theta \to \theta$ |
| $\mathrm{fix}_\theta$ | : | $(\theta \to \theta) \to \theta$ |
| $\mathrm{Var}[\delta]$ | : | $\mathsf{cls}\,\mathsf{var}[\delta]$ |
| $\_ *_{\theta_1,\theta_2} \_$ | : | $\mathsf{cls}\,\theta_1 \times \mathsf{cls}\,\theta_2 \to \mathsf{cls}\,(\theta_1 \times \theta_2)$ |

**TABLE 3**
**Essential constants of IA$^+$**

There are no restrictions on what free identifiers can occur in a class term. So, it is possible for the **meth** term to modify non-local variables. It is also possible for the initialization command to modify non-local variables.

The important constants of IA$^+$ are shown in Table 3. (The constants for expression and value types are omitted.) The constant $\mathsf{skip}$ denotes the do-nothing command and ";" denotes sequential composition. The **letval** operator sequences the evaluation of an expression with that of another expression or command. More precisely, **letval** $e\ f$ evaluates $e$ in the current state to obtain a value $x$ and then evaluates $f\ x$. (Note that this would not make sense if **letval** $e\ f$ were of type $\mathsf{val}[\delta']$.) In typical usage, **letval** is used to evaluate an expression and bind its value to an identifier, e.g.,

$$\begin{array}{l}\mathbf{letval}\ e\ \lambda x.\\ \quad A(x)\end{array}$$

The **letval** primitive provides a mechanism for forcing the evaluation of an expression inside another expression or a command. Such forcing cannot be done in all types of values. For example, values of type $\mathsf{val}[\delta]$ are static and state-independent. So, they cannot incorporate an expression evaluation. We identify a class of types called "hereditarily state-dependent types" which support the forcing of expression evaluation. They are given by the following syntax:

$$\sigma \quad := \quad \mathsf{exp}[\delta] \mid \mathsf{comm} \mid \sigma_1 \times \sigma_2 \mid \{x_i\colon \sigma_i\}_i \mid \theta_1 \to \sigma_2$$

Note that types of the form $\mathsf{val}[\delta]$ and $\mathsf{cls}\,\theta$ are not hereditarily state-dependent. The **letval** operator is extended to hereditarily state-dependent types as follows:

$$\begin{array}{l}\mathbf{letval}_{\delta,\sigma_1\times\sigma_2}\ e\ f\\ \quad = (\mathbf{letval}_{\delta,\sigma_1}\ e\ (fst \circ f), \mathbf{letval}_{\delta,\sigma_2}\ e\ (snd \circ f))\\ \mathbf{letval}_{\delta,\{x_i:\sigma_i\}_i}\ e\ f\\ \quad = \{x_i = \mathbf{letval}_{\delta,\sigma_i}\ e\ \lambda k.\ (f\ k).x_i\}_i\\ \mathbf{letval}_{\delta,\theta_1\to\sigma_2}\ e\ f\\ \quad = \lambda x\colon \theta_1.\ \mathbf{letval}_{\delta,\sigma_2}\ e\ \lambda k.\ f\ k\ x\end{array}$$

Since values of all hereditarily state-dependent types are eventually used in the context of a state, they can incorporate expression evaluation as a component. Values of other types do not have this capability.

*Call by value.* We also use an implicit conversion that corresponds to Algol's notion of call by value. If $f : \mathsf{val}[\delta] \to \sigma$ is a value-accepting function to a hereditarily state-dependent type $\sigma$, and $e : \mathsf{exp}[\delta]$ is an expression, we allow an application of the form $(f\ e)$ with the interpretation:

$$f\ e \quad = \quad \mathbf{letval}\ e\ \lambda x.\ f(x)$$

We call this "call-by-value application." Notice its use in writing self.set(self.val + 1) in the *SetCounter* class above. It is also used in writing typical conditional commands of the form

$$\mathbf{if}\ E\ A\ B$$

where $E$ is a state-dependent expression of type $\mathsf{exp}[\mathsf{bool}]$. The implicit call-by-value application has the effect that type declarations become mandatory. For example, the function abstraction term:

$\lambda$x. (y := y + 1; print x)

can be assigned both the types of the form $\mathsf{val}[\delta] \to \mathsf{comm}$ and $\mathsf{exp}[\delta] \to \mathsf{comm}$ with quite different meanings. (Consider applying the function to $y$.)

The infix operator ":=" for variable assignment is defined by:

$$\begin{array}{l} \text{":="} : \mathsf{var}[\delta] \times \mathsf{exp}[\delta] \to \mathsf{comm} \\ v := e \quad \stackrel{\text{def}}{=} \quad \mathbf{letval}\ e\ \lambda x.\ (v.\mathrm{put}(x)) \end{array}$$

Note that it forces expression evaluation via **letval**.

An important property of all the constants mentioned in Table 3 is that they do not have global side effects.[5] This is a requirement of our semantics, imposed to ensure that closed terms are free of global side effects. The property would be violated, for instance, if we were to add a constant $\mathbf{print} : \mathsf{val}[\delta] \to \mathsf{comm}$ for printing. (A closed term, like **print**(20), would cause a global state change.) The preferred method is to treat *print* as a free identifier that is bound in the environment of program execution.

*Equational properties*

The equational calculus for the typed lambda calculus part of IA$^+$ is standard. For cls type constructs, we have the following laws:

$$\begin{array}{ll} (\beta) & \mathbf{new}\ (\mathbf{class}:\ \theta\ \mathbf{local}\ C\ x\ \mathbf{init}\ A\ \mathbf{meth}\ M) \\ & \qquad = \lambda p.\ \mathbf{new}\ C\ \lambda x.\ A; p\ M \\ (\eta) & (\mathbf{class}:\ \theta\ \mathbf{local}\ C\ x\ \mathbf{init}\ \mathsf{skip}\ \mathbf{meth}\ x) \\ & \qquad = C \end{array}$$

[5] A function-typed value in an Algol-like language is said to have a "side effect" if it involves state changes other than those of its arguments.

The ($\beta$) law specifies the effect of an Intro-Elim combination. The ($\eta$) law specifies the effect of an Elim-Intro combination where the "Elim" is the implicit elimination in local object declarations.

The **new** operator supports a number of interesting equational properties. Unfortunately, these properties do not hold in general because the initialization command of a class may have global side effects (change objects other than the local objects of the class). However, most classes used in practice are defined by closed terms. We call such classes "closed classes." Since closed terms do not have global side effects, the properties of interest hold for them. These properties also hold for a more general class of terms called "constant terms" defined in the Appendix.

The following equation scheme allows one to reorder **new** declarations. Whenever $C_1$ and $C_2$ are closed classes:

$$\mathbf{new}\, C_1\, \lambda x.\, \mathbf{new}\, C_2\, \lambda y.\, M \;=\; \mathbf{new}\, C_2\, \lambda y.\, \mathbf{new}\, C_1\, \lambda x.\, M \tag{2}$$

The interaction of **new** declarations with various constants is expressed by the following equation schemes (where $C$ is a closed class and $a, b : \mathsf{comm}$, $f, g : \theta \to \mathsf{comm}$, $e : \mathsf{exp}[\delta]$, $h : \theta \to \mathsf{val}[\delta] \to \mathsf{comm}$ and $p : \mathsf{val}[\mathsf{bool}]$ are free identifiers):[6]

$$\mathbf{new}\, C\, \lambda x.\, \mathsf{skip} \;=\; \mathsf{skip} \tag{3}$$

$$\mathbf{new}\, C\, \lambda x.\, (a; g(x)) \;=\; a;\, \mathbf{new}\, C\, \lambda x.\, g(x) \tag{4}$$

$$\mathbf{new}\, C\, \lambda x.\, (g(x); b) \;=\; (\mathbf{new}\, C\, \lambda x.\, g(x));\, b \tag{5}$$

$$\left\{\begin{array}{l}\mathbf{new}\, C\, \lambda x. \\ \quad \mathbf{letval}\, e\, \lambda z.\, h\, x\, z\end{array}\right\} \;=\; \left\{\begin{array}{l}\mathbf{letval}\, e\, \lambda z. \\ \quad \mathbf{new}\, C\, \lambda x.\, h\, x\, z\end{array}\right\} \tag{6}$$

$$\mathbf{new}\, C\, \lambda x.\, \mathbf{if}\, p\, (f\, x)\, (g\, x) \;=\; \mathbf{if}\, p\, (\mathbf{new}\, C\, f)\, (\mathbf{new}\, C\, g) \tag{7}$$

(In the presence of nonterminating initializations, the equation (3) must be weakened to an inequality $\mathbf{new}\, C\, \lambda x.\, \mathsf{skip} \sqsubseteq \mathsf{skip}$. We are also ignoring the issue of "visible effects," such as printing, which might occur before nontermination and invalidate equations like (2) and (4).) These equations state that the **new** operator commutes with all the operations of IA$^+$. Any computation that is independent of the new instance can be moved out of the scope of **new**. Compilers (implicitly) use these kinds of equations to enlarge or contract the scope of local variables and to eliminate "dead" variables. By formally introducing classes as a feature, we are able to generalize them from variables to all objects. Notice that, by setting $g = \lambda x.\, \mathsf{skip}$ in (4) and using (3), we can derive the famous equation:

$$\mathbf{new}\, C\, \lambda x.\, a \;=\; a \tag{8}$$

This equivalence has been discussed in various papers on semantics of local variables [38, 39, 50].

[6] Note that these are equations of the typed lambda calculus. The symbols $a, g, \ldots$ are *free identifiers* which can never be substituted by terms that capture bound identifiers. For instance, in equation (4), $a$ cannot be substituted by a term that has $x$ occurring free.

In [59, Appendix], Reynolds suggests encoding classes as their corresponding "new" operators. This involves the translation:

$$\mathsf{cls}\,\theta \;\rightsquigarrow\; (\theta \to \mathsf{comm}) \to \mathsf{comm}$$

$$(\mathbf{class}:\ \theta\ \mathbf{local}\, C\ x\ \mathbf{init}\, A\ \mathbf{meth}\, M) \;\rightsquigarrow\; \lambda p.\,\mathrm{new}\, C\, \lambda x.\,(A; p(M))$$

$$\mathbf{new}\, C \;\rightsquigarrow\; C$$

For instance, the class *Counter* would be encoded as an operator $newCounter : (counter \to \mathsf{comm}) \to \mathsf{comm}$. Unfortunately, arbitrary functions of this type do not satisfy the axioms of **new** listed above. The reason is that the type of *newCounter* does not constrain it to call its argument procedure exactly once. (This means that Reynolds's encoding does not give a fully abstract translation from $\mathrm{IA}^+$ to Idealized Algol.) Our treatment can be seen as a formalization of the properties intrinsic to "new" operators of classes.

## 4. SPECIFICATIONS

An ideal framework for specifying classes in $\mathrm{IA}^+$ is the *specification logic* of Reynolds [61]. (See the survey article [67] for a detailed description of specification logic.) Specification logic can be regarded as a theory within (typed) first-order intuitionistic logic. We use the following intuitionistic connectives:

| | |
|---|---|
| $\&$ | conjunction |
| $\Longrightarrow$ | implication |
| $\forall$ | universal quantification |
| $\exists$ | existential quantification |

The types include those of Idealized Algol and an additional base type $\mathsf{assert}$ for assertions (state-dependent classical logic formulas). The atomic formulas of specification logic include:

- equations, $M =_\theta N$, for $\theta$-typed terms $M$ and $N$,
- Hoare-style partial correctness triples, $\{P\}\,A\,\{Q\}$, for command $A$ and assertions $P$ and $Q$, and
- non-interference formulas, $A \#_{\theta,\theta'} B$, where $A$ and $B$ are terms of types $\theta$ and $\theta'$ respectively.

The type rules for these formulas are shown in Table 4 using judgments of the form "$\varphi$ Formula." Note that assertions form a "logic within logic." One can use classical reasoning for them even though the outer logic is intuitionistic. Specification logic includes fixed point induction to deal with recursion. This is typically used for proving partial correctness properties. Termination must be proved separately (outside the logic).

*Non-interference*

A non-interference formula $A \# B$ (read "$A$ does not interfere with $B$" or "$A$ is independent of $B$") means intuitively that $A$ and $B$ do not access any common storage locations except in a read-only fashion. We use a *symmetric* non-interference

$$\frac{\Gamma \rhd M : \theta \quad \Gamma \rhd N : \theta}{\Gamma \rhd M =_\theta N \text{ Formula}} \qquad \frac{\Gamma \rhd M : \theta \quad \Gamma \rhd N : \theta'}{\Gamma \rhd M \#_{\theta,\theta'} N \text{ Formula}}$$

$$\frac{\Gamma \rhd P : \mathsf{assert} \quad \Gamma \rhd A : \mathsf{comm} \quad \Gamma \rhd Q : \mathsf{assert}}{\Gamma \rhd \{P\}\, A\, \{Q\} \text{ Formula}}$$

$$\frac{\Gamma \rhd C : \mathsf{cls}\,\theta \quad \Gamma, x{:}\,\theta \rhd \varphi \text{ Formula}}{\Gamma \rhd \mathbf{Inst}\; C\; x.\, \varphi \text{ Formula}}$$

**TABLE 4**
**Selected type rules of IA$^+$ specification logic**

predicate (from [59, 47]), which is somewhat easier to use than Reynolds's original version in specification logic. The basic facts for non-interference come from instance declarations. A newly created instance is independent of all other existing objects, unless its class interferes with those objects. Starting from these facts, we can infer non-interference for more complex terms using the following proof rules:

1. If $A$ and $B$ are terms with free identifiers $\{x_i\}_i$ and $\{y_j\}_j$ then

$$\&_{i,j}(x_i \# y_j) \Longrightarrow A \# B$$

2. If both $A$ and $B$ are of "passive" types then $A \# B$.
3. If either $A$ or $B$ is of a "constant" type then $A \# B$.
4. If $A \# B$ then $B \# A$.

Passive types are those that hereditarily lead to $\mathsf{val}[\delta]$ or $\mathsf{exp}[\delta]$ types, and constant types are those that hereditarily lead to $\mathsf{val}[\delta]$. They are given by the following syntax:

$$\begin{array}{lll} \text{(Passive types)} & \phi & ::= \ \mathsf{val}[\delta] \mid \mathsf{exp}[\delta] \mid \phi_1 \times \phi_2 \mid \theta_1 \to \phi_2 \mid \{x_i{:}\,\phi_i\}_i \\ \text{(Constant types)} & \tau & ::= \ \mathsf{val}[\delta] \mid \tau_1 \times \tau_2 \mid \theta_1 \to \tau_2 \mid \{x_i{:}\,\tau_i\}_i \end{array}$$

See Appendix for further discussion, where there are also new typing mechanisms defined for enlarging these classes in a significant way.

The first rule of non-interference reduces the non-interference of terms to that of their free identifiers. If all the free identifiers of $A$ and $B$ are non-interfering, then $A$ and $B$ are non-interfering. Passive types, used in the second rule, identify computations that only read the state. Two computations that only read the state are always non-interfering. In the third rule, constant types identify computations that neither read nor write the state. Such computations do not interfere with anything.

The effect of the non-interference predicate is best illustrated by the axiom:

$$\forall a, b : \mathsf{comm}. \quad a \# b \Longrightarrow a; b = b; a$$

which states that two non-interfering commands can be freely reordered. The equivalences stated in Sec. 3 can also be formalized as axioms using the non-interference predicate. For example, the equivalence (2) can be stated as:

$$\begin{array}{l}\forall c_1\colon \mathsf{cls}\,\theta_1.\,\forall c_2\colon \mathsf{cls}\,\theta_2.\,\forall g\colon \theta_1 \times \theta_2 \to \mathsf{comm}.\; c_1 \,\#\, c_2 \Longrightarrow \\ \quad \mathbf{new}\; c_1\; \lambda x.\; \mathbf{new}\; c_2\; \lambda y.\; g(x,y) \\ \quad\quad = \mathbf{new}\; c_2\; \lambda y.\; \mathbf{new}\; c_1\; \lambda x.\; g(x,y)\end{array}$$

*Class specifications*

For handling IA$^+$, we extend specification logic with cls types and add a new formula of the form:

$$\mathbf{Inst}\; C\; x.\; \varphi(x)$$

where $C$ is a class, $x$ an identifier (bound in the formula) and $\varphi(x)$ is a formula. The meaning is that all instances $x$ of class $C$ satisfy the formula $\varphi(x)$. An example is the following specification of the variable class:

**Inst** Var[$\delta$] x.
  $\forall$p: exp[$\delta$] $\to$ assert. x # p $\Longrightarrow$
    {p(k)} x.put k {p(x.get)}

Thus, the Hoare logic's axiom scheme for assignment becomes a specification of the variable class.

One can also write equational specifications for classes. For example, consider the specification of counters given by:

**Inst** Counter x.
  $\forall$g: exp[int] $\to$ comm. x # g $\Longrightarrow$
    x.inc; g(x.val) = g(x.val + 1); x.inc

The quantified function identifier $g$ plays the role of a "conversion" function, to convert expressions into commands. The specification says that incrementing the counter and using its value in some context $g$ is equivalent to using one plus the value before incrementing the counter. In essence, the effect of *inc* is to increment the *val* of the counter. As a less trivial example, an equational specification of a *Queue* class is shown in Table 5. Its structure is similar to that of the *Counter* specification.

Specification logic allows the use of both equational reasoning and reasoning via Hoare-triples. The choice between them is a matter of preference, but Hoare-like reasoning is better understood and is often simpler. For example, a Hoare-triple specification of counters can be written as

**Inst** Counter x.
  $\forall$k: val[int].
    {x.val = k} x.inc {x.val = k + 1}

This states much more directly that the effect of $x.inc$ is to increment $x.val$.

For more interesting data structures, where the state is not directly accessible via methods, Hoare-triple specifications can be written using abstraction predicates.

**TABLE 5**
**Equational specification of a queue class**

```
type queue = {init: comm, ins: val[int] → comm, del: comm, front: exp[int] }
Queue : cls queue
Inst Queue q.
  ∀x,y: val[int]. ∀g: exp[int] → comm. g # q ⟹
       q.init; q.ins(x); q.del   =   q.init
  &    q.ins(x); q.ins(y); q.del   =   q.ins(x); q.del; q.ins(y)
  &    q.init; q.ins(x); g(q.front)   =   q.init; q.ins(x); g(x)
  &    q.ins(x); q.ins(y); g(q.front)   =   q.ins(x); g(q.front); q.ins(y)
```

**TABLE 6**
**Hoare-triple specification of queues**

```
Inst Queue q.
  ∃elems: val[list int] → assert.
  ∀k: val[int]. ∀s: val[list int].
       {true} q.init {elems([ ])}
  &    {elems(s)} q.ins(k) {elems(s@[k])}
  &    {elems(k::s)} q.del {elems(s)}
  &    {true} skip {elems(k::s) ⇒ q.front = k}
```

For example, in Table 6, we show a Hoare-triple specification of *Queue*. The specification asserts the existence of an *elems* predicate representing an abstraction of the internal state of the queue as a list. (We are using an ML-like notation for lists. Note also that we are regarding list int as a data type for the purpose of abstract reasoning.) The logical facilities of specification logic allow us to specify the *existence* of an abstraction function whose definition can only be determined in the context of an implementation of the class.

Consider an implementation of the *Queue* class using "unbounded" arrays,[7] shown in Table 7. To show that it meets the Hoare-triple specification, we pick the assertion:

$$\text{elems(s)} \iff \text{f} \leq \text{r} \wedge \text{a[f} + 1, \ldots, \text{r]} = \text{s}$$

A *Queue*-state represents a queue with elements $s$ iff $f \leq r$ and the list of array elements between $f+1$ and $r$ is $s$. (We are using the notation $a[f+1, \ldots, r]$ for the array section between the two bounds, regarded as a list.) Note that the predicate incorporates both the "representation invariant"(the condition $f \leq r$) and the

[7] We are using "unbounded" arrays as an abstraction to finesse the technicalities of bounds. Clearly, both the specification and the implementation of *Queue* can be modified to deal with bounded queues.

**TABLE 7**
**An implementation of queues**

```
Queue =
  class queue
    local (UnboundedArray Var[int]) a;
          Var[int] f, r
    init (f := 0; r := 0)
    meth
        {init = (f := 0; r := 0),
         ins = λx. (r := r + 1; a(r) := x),
         del = (if f < r then f := f + 1 else skip),
         front = if f ≠ r then a(f + 1).get else 0 }
```

"representation function" (given by the expression $a[f+1,\ldots,r]$) in conventional terminology [6].

Specification logic is also able to express "history properties" of the kind recommended by Liskov and Wing [36]. For example, here is a formula that states that a counter's value can only increase over time:

**Inst** Counter x.
  ∀k: val[int]. ∀a: comm.
    {x.val = k} a {x.val ≥ k}

Note that we do not have an assumption $x \# a$ in this specification. So, it is possible for $a$ to make its own state changes to $x$ (through aliasing, for example). The specification still holds because the only possible state changes to $x$ are via the *inc* operation. On the other hand, if we were to replace *Counter* by *SetCounter*, the specification would fail. In that case, the command $a$ can potentially decrease $x$ through the *set* operation. In general, adding methods to a class can falsify its history properties.

Using **Inst**-specifications, we formulate the following proof rule for **new** declarations:[8]

$$\frac{\begin{array}{c}\left[\begin{array}{c}\varphi(x)\\ \{C \# T_i \Longrightarrow x \# T_i\}_i\end{array}\right]\\ \vdots\\ \textbf{Inst}\ C\ x.\,\varphi(x) \qquad \{P\}\ g\ x\ \{Q\}\end{array}}{\{P\}\ \textbf{new}\ C\ g\ \{Q\}} \tag{9}$$

where $x$ does not occur free in any undischarged assumptions, the terms $T_i$ and the assertions $P$ and $Q$. This states that, to prove a Hoare triple specification for (**new** $C$ $g$), we need to prove it for ($g$ $x$), where $x$ is an arbitrary instance of

[8] We are presenting the rules in a natural deduction form for readability. They can also be stated as axioms in the style of [61].

$C$. During the proof, we get to assume that $x$ satisfies the specification $\varphi(x)$ and the fact that $x$ does not interfere with anything unless $C$ interferes with it. The terms $T_i$ can be any terms whatever but, in a typical usage of the rule, they are the free identifiers of the specification $\{P\}\, g\, x\, \{Q\}$. These non-interference assumptions arise from the fact that $x$ is a "new" instance. They form the basic raw material for non-interference reasoning.

The rule for inferring **Inst**-specifications is:

$$\frac{\mathbf{Inst}\ C\ z.\ \psi(z) \qquad \begin{bmatrix} \psi(z) \\ \{C\ \#\ T_i \Longrightarrow z\ \#\ T_i\}_i \end{bmatrix} \\ \vdots \\ \varphi(M)}{\mathbf{Inst}\ (\mathbf{class}:\ \theta\ \mathbf{local}\ C\ z\ \mathbf{init}\ A\ \mathbf{meth}\ M)\ x.\ \varphi(x)} \tag{10}$$

where $z$ does not occur free in any undischarged assumptions, the terms $T_i$ and the formula $\varphi(-)$. The proof that the queue implementation of Table 7 satisfies the Hoare-triple specification is carried out using this rule. Proving that the queue implementation satisfies to the equational specification of Table 5 is more involved. We discuss it in Sec. 5.1.

The initialization command $A$ does not play any role in the above proof rule because **Inst**-specifications state the properties that hold in *all* states, not only the initial state. To state the properties that hold in the initial state, axioms involving **new**-terms can be used. For example, the *Counter* class satisfies the following "initialization" axiom:

$$\left\{\begin{array}{c}\mathbf{new}\ \text{Counter}\ \lambda\text{x.} \\ \text{g(x.val); h(x)}\end{array}\right\} \quad = \quad \left\{\begin{array}{c}\mathbf{new}\ \text{Counter}\ \lambda\text{x.} \\ \text{g(0); h(x)}\end{array}\right\}$$

which specifies that the initial value of a newly created counter is 0. Such initial value axioms are a bit cumbersome to write because they have to specify equalities that hold in a particular context. The properties specified in **Inst**-specifications, on the other hand, hold in all contexts.

No new logical principles are involved in handling self-reference and inheritance because these concepts are modelled using recursion. For example, the proof principle for self-referential classes can be derived from fixed-point induction:

$$\begin{array}{c}\varphi(\bot) \wedge (\mathbf{Inst}\ c\ f.\ \forall x.\ \varphi(x) \Longrightarrow \varphi(f(x))) \\ \Longrightarrow \mathbf{Inst}\ (\text{close}\ c)\ x.\ \varphi(x)\end{array}$$

So, to verify that the instances of (*close* $c$) satisfy $\varphi$, we need to show that the instances of $c$ preserve $\varphi$. As an example, the *SetCounter* class can be shown to satisfy:

$$\begin{array}{l}\mathbf{Inst}\ \text{SetCounter f. } \forall\text{x. } \varphi(x) \Longrightarrow \varphi(f(x)) \\ \text{where } \varphi(x) \equiv\ \forall\text{k: val[int]. } \forall\text{p: exp[int]} \rightarrow \text{assert. x } \#\ \text{p} \Longrightarrow \\ \qquad\qquad \{\text{p(k)}\}\ \text{x.set k } \{\text{p(x.val)}\} \\ \qquad\qquad \&\ \{\text{p(x.val + 1)}\}\ \text{x.inc } \{\text{p(x.val)}\}\end{array}$$

Since the formula $\varphi(x)$ is a partial-correctness specification, it trivially holds for $\bot$. Hence, we have

**Inst** (close SetCounter) x. $\varphi$(x)

See [30] for more discussion of this and other similar techniques.

The non-interference conditions occurring at various parts of this theory might seem unusual and somewhat heavy but, once their role is understood, they are quite easy to handle and are seen to help reasoning considerably. The first proof rule of non-interference reduces non-interference of terms to non-interference of their free identifiers: two terms $A$ and $B$ are non-interfering if all the free identifiers of $A$ are non-interfering with all the free identifiers of $B$. This is easily ensured by a syntactic examination of $A$ and $B$, *provided* we know which free identifiers have the possibility of interference. It is usually a good practice to make sure that no two free identifiers of a term or formula interfere. In our example formulas, we followed this practice. For example, in the equational specification of *Counter*, we laid down the condition $x \# g$ as soon as the two identifiers are introduced in the context. Reynolds has also defined a system for "Syntactic Control of Interference" [59] where it can be automatically verified that no two free identifiers interfere. If this practice is strictly followed, then the non-interference of $A$ and $B$ can be ensured by just checking that they have no common free identifiers. (The second and third axioms relax this condition by allowing certain kinds of free identifiers to be shared by $A$ and $B$.)

Within programs, the basic raw material for showing non-interference comes from instance declarations. Whenever a new instance of a class $C$ is created, it is known to be non-interfering with anything that $C$ does not interfere with. Since most classes are defined by closed terms (e.g., the *Queue* class of Table 7), such classes do not interfere with anything. If a class is given by a closed term, evey instance of the class is non-interfering with other objects previously in existence.

Thus, though non-interference conditions seem to have an overbearing presence in the theory, reasoning about them is usually straightforward in most practical situations. An exception to this observation is the handling of data structures, e.g., arrays. If we want to pass two array components, say, $a(i)$ and $a(j)$, as arguments to a procedure and the procedure specification requires the two arguments to be non-interfering, we have to reason about the inequality of $i$ and $j$. Techniques for such reasoning are still under investigation.

## 5. SEMANTICS

The denotational semantics of IA$^+$ brings out important properties of classes and objects. We consider two styles of semantics: *parametricity* semantics along the lines of [51], which highlights the data abstraction aspects of classes, and *object-based* semantics along the lines of [58], which highlights the class-instance relationship.

### 5.1. Parametricity semantics

Recall, from Sec. 2, that objects can be regarded as state machines with a state set $Q$ and operations that act on the state set $Q$. Since these operations correspond to methods written in IA$^+$, it follows that IA$^+$ types $\theta$ correspond to *type constructors* parameterized by state sets $Q$. For example, methods of type comm are interpreted as state transformations of type $(Q \to Q)$. So, corresponding to the IA$^+$ type

$\mathsf{comm}$, we have the type constructor $(- \to -)$ which maps any state set $Q$ to the set of state transformations for $Q$. Similarly, for every IA$^+$ type $\theta$, we have a type constructor $[\![\theta]\!]$ so that a method of type $\theta$ in an object with state set $Q$ can be interpreted as an operation of type $[\![\theta]\!](Q)$. The interpretation is as follows:

$$\begin{aligned}
[\![\mathsf{exp}[\delta]]\!](Q) &= Q \to [\![\delta]\!] \\
[\![\mathsf{comm}]\!](Q) &= Q \to Q \\
[\![\mathsf{val}[\delta]]\!](Q) &= [\![\delta]\!] \\
[\![\theta_1 \times \theta_2]\!](Q) &= [\![\theta_1]\!](Q) \times [\![\theta_2]\!](Q) \\
[\![\{x_i\colon \theta_i\}_i]\!](Q) &= \textstyle\prod_{x_i} [\![\theta_i]\!](Q) \\
[\![\theta_1 \to \theta_2]\!](Q) &= \forall Z.\, [\![\theta_1]\!](Q \times Z) \to [\![\theta_2]\!](Q \times Z) \\
[\![\mathsf{cls}\,\theta]\!](Q) &= \exists Z.\, [\![\theta]\!](Q \times Z) \times [Q \to Q \times Z]
\end{aligned} \tag{11}$$

In interpreting types using sets, we are ignoring the issues of termination and recursion. See the end of this section for remarks on how to extend it to handle these features.

Expressions are interpreted as functions from states to values (modeling state-dependent valuations) and commands as functions from states to states (modeling state transformations). Value types are simply interpreted as sets of values. We are using $[\![\delta]\!]$ to mean the set of values for the data type $\delta$ ($[\![\mathsf{int}]\!]$ is the set of integers, $[\![\mathsf{bool}]\!]$ is the set of boolean values, etc.) Product types are interpreted as pointwise products, because a pair of methods corresponds to a pair of operations. Record types are similarly interpreted as pointwise products (indexed by field identifiers). We will ignore record types in the remainder of this section because they are very much similar to product types. The interpretation of function types and class types is more sophisticated.

When a function method of type $\theta_1 \to \theta_2$ is called, we can supply an argument which is potentially dependent on some other object. Thus, the argument lives in an expanded state set $Q \times Z$ which incorporates both the state set of the receiver object and the extra state of the argument object. The result of the method-call likewise lives in the expanded state set. Moreover, the method must be prepared to accept arguments in all possible expanded state sets, treating those expansions in a uniform way. This explains the quantification $\forall Z$ in the interpretation of function types. This is the same form quantification as in polymorphic lambda calculus [62]. (See also [42, Ch. 9].)

The interpretation of class-types involves the dual form of quantification $\exists Z$. A class defined in the context of some state set $Q$, first specifies a state set $Z$ for the objects of the class. In addition, it gives a method suite of type $\theta$ which acts on the combined state set $Q \times Z$ of the context and the object and, finally, an initialization operation (of type $Q \to Q \times Z$). The quantification involved in this interpretation is existential quantification because the class definition provides a state set $Z$ which serves as a hypothetical state set for describing the behavior of the class. This form of existential quantification was introduced by Mitchell and Plotkin [43] to describe the types of data abstractions. See also Cardelli and Wegner [14] and [42, Ch. 9] for a detailed discussion of existential quantification.

An IA$^+$ term with typing $x_1 : \theta_1, \ldots, x_n : \theta_n \rhd M : \theta$ is interpreted as a polymorphic function of type

$$[\![M]\!] : \forall Q.\, [\![\{x_1 : \theta_1, \ldots, x_n : \theta_n\}]\!](Q) \to [\![\theta]\!](Q)$$

Thus, the term $M$ has a value in every state set $Q$ in which the free identifiers can be assigned values. And, this interpretation is "uniform," i.e., acts the same way for all state sets. The fact that the meanings of terms are polymorphic functions, not ordinary functions, leads to a characteristic difference between closed terms and open terms. Possible values for closed terms are often few. For example, the only possible values for closed terms of type comm are the diverging command and skip. On the other hand, open terms of type comm have more interesting possibilities because they can use the state information of the free identifiers.

To formalize the behavioral equivalence of classes as well as the uniformity of IA$^+$ functions, we must interpret the type expressions occurring in (11) using the ideas of relational parametricity [62, 51]. The idea is that every type expression $T(-)$ denotes a *type operator* which not only maps each state set $Q$ to a set $T(Q)$, but also maps every relation $R : Q \leftrightarrow Q'$ between state sets to a relation $T(R) : T(Q) \leftrightarrow T(Q')$. The meaning of the quantifiers $\forall$ and $\exists$ take this relational action into account. The basic ideas for this interpretation are due to O'Hearn and Tennent [51] and we follow the presentation in Section 2 of their paper. In particular, we ignore recursion and curried functions. The later discussion in [51] about handling these features is immediately applicable.

*Type operators*

A *unary type operator of* $T$ over a collection of sets $\mathcal{S}$ is a pair $\langle T_{\rm set}, T_{\rm rel}\rangle$ where

- the "set part" $T_{\rm set}$ assigns to each set $X \in \mathcal{S}$, a set $T_{\rm set}(X)$, and
- the "relation part" $T_{\rm rel}$ assigns to each binary relation $R : X \leftrightarrow X'$, a relation $T_{\rm rel}(R) : T_{\rm set}(X) \leftrightarrow T_{\rm set}(X')$.

such that $T_{\rm rel}(\Delta_X) = \Delta_{T_{\rm set}(X)}$ where $\Delta_X$ denotes the identity relation of $X$. (We normally write both $T_{\rm set}$ and $T_{\rm rel}$ as simply $T$, using the context to disambiguate the notation.) The condition that $T(\Delta_X) = \Delta_{T(X)}$ is called the "identity extension" property. One can define $n$-ary type operators similarly, with set parts of the form $T(X_1, \ldots, X_n)$ and relation parts of the form $T(R_1, \ldots, R_n)$. The identity extension property is $T(\Delta_{X_1}, \ldots, \Delta_{X_n}) = \Delta_{T(X_1,\ldots,X_n)}$. We use type operators of this kind to interpret type expressions occurring in the interpretation (11).

Since our type operators involve quantifiers, we assume that the collection $\mathcal{S}$ forming the range of type variables is a set. Note that, in our application, $\mathcal{S}$ is the collection of state sets. For most practical purposes, $\mathcal{S}$ can be taken to be the set of countable sets. Alternatively, one can assume a *universe* set that is closed under all set-theoretic constructions [37, Sec. I.6] .

We have the following basic type operators:

$$\begin{array}{lrcl}
\text{Identity} & \mathrm{J}(X) & = & X \\
 & \mathrm{J}(R) & = & R \\
\text{Constant} & \bar{A}(X) & = & A \quad \text{(for a set A)} \\
 & \bar{A}(R) & = & \Delta_A
\end{array}$$

For the $n$-ary case, we also have the projection operators $\pi_i^n$ with $\pi_i^n(\vec{X}) = X_i$ and $\pi_i^n(\vec{R}) = R_i$ . The product and function-space constructions have their counterparts for type operators:

$$\begin{array}{lrcl} \text{Product} & (T_1 \times T_2)(X) & = & T_1(X) \times T_2(X) \\ & (T_1 \times T_2)(R) & = & T_1(R) \times T_2(R) \\ \text{Function space} & (T_1 \to T_2)(X) & = & T_1(X) \to T_2(X) \\ & (T_1 \to T_2)(R) & = & T_1(R) \to T_2(R) \end{array}$$

The relation operators $\times$ and $\to$ used here are standard:

$$\begin{array}{rcl} (x, y)\ [R \times S]\ (x', y') & \iff & x\ R\ x' \wedge y\ S\ y' \\ f\ [R \to S]\ f' & \iff & \forall x, x'.\, x\ R\ x' \Longrightarrow f(x)\ S\ f'(x') \end{array}$$

For readability, we often denote type operators by type expressions. For example, the type expression

$$T(X) \ = \ (X \to A) \times (X \to B) \to (X \to A \times B)$$

in a type variable $X$ denotes the type operator:

$$T \ = \ (\mathrm{J} \to \overline{A}) \times (\mathrm{J} \to \overline{B}) \to (\mathrm{J} \to \overline{A \times B})$$

The relation part of the type operator is:

$$T(R) \ = \ (R \to \Delta_A) \times (R \to \Delta_B) \to (R \to \Delta_{A \times B})$$

whose form parallels that of $T(X)$ except that the set variable $X$ is replaced by a relation variable $R$ and the set constants $A$, $B$ and $A \times B$ are replaced by their identity relations. (This is generally the case.)

*Quantified type operators*

Next, we define quantifiers for type operators. The universal quantifier $\forall$ represents parametrically polymorphic functions and the existential quantifier $\exists$ represents abstract data types.

If $T(X, Z)$ is a binary type operator, we have a unary type operator $\forall Z.\, T(X, Z)$ which represents parametrically polymorphic functions $p$ with components $p_Z \in T(X, Z)$ for each set $Z \in \mathcal{S}$. (The "component" $p_Z$ is nothing but the instance of the polymorphic function at type $Z$. We also use the notation $p[Z]$ to denote such a component.) The type operator $\forall^1(T)$ (denoted informally by the type expression $\forall Z.\, T(X, Z)$) is defined as follows:

- the set part maps a set $X$ to the set $\forall Z.\, T(X, Z)$ whose elements are $\mathcal{S}$-indexed families $p = \{p_Z\}_{Z \in \mathcal{S}}$ such that, for all relations $S : Z \leftrightarrow Z'$,

$$p_Z\ [T(\Delta_X, S)]\ p_{Z'}$$

- the relation part maps a relation $R : X \leftrightarrow X'$ to the relation $\forall S.\, T(R, S) : \forall Z.\, T(X, Z) \leftrightarrow \forall Z.\, T(X', Z)$ defined by

$$\begin{array}{l} p\ [\forall S.\, T(R, S)]\ p' \iff \\ \qquad \forall Z, Z' \in \mathcal{S}.\, \forall S : Z \leftrightarrow Z'.\, p_Z\ [T(R, S)]\ p'_{Z'} \end{array}$$

(Similarly, one can define $\forall^n(T)$ for each arity $n \geq 0$.) The condition $p_Z\ [P(\Delta_X, S)]\ p_{Z'}$ in the set part is referred to as the "parametricity condition." It ensures that all the components $p_Z$ of the polymorphic function act the same way. (The identity relation $\Delta_X$ occurs in the first argument position because all the components $p_Z$ are defined for the same set $X$ in the first position.) As an example, consider the family of functions

$$\begin{aligned} &swap_X \in \forall Z.\ X \times Z \to Z \times X \\ &(swap_X)_Z(x, z) \ = \ (z, x) \end{aligned}$$

We verify that this is parametrically polymorphic by noting that $(swap_X)_Z$ and $(swap_X)_{Z'}$ are related by $[\Delta_X \times S \to S \times \Delta_X]$ , i.e.,

$$(x, z)\ [\Delta_X \times S]\ (x', z') \Longrightarrow (z, x)\ [S \times \Delta_X]\ (z', x')$$

This property is often denoted diagramatically by:

$$\begin{array}{ccc} X \times Z & \xrightarrow{(swap_X)_Z} & Z \times X \\ \Delta_X \times S \Big\updownarrow & & \Big\updownarrow S \times \Delta_X \\ X \times Z' & \xrightarrow{(swap_X)_{Z'}} & Z' \times X \end{array}$$

The intuition is that the $swap_X$ family is uniform: it acts the same way for every type $Z$. If $swap_X$ were non-uniform, for example, by negating the second component of the pair for $Z = Bool$ and leaving it unchanged for all other $Z$, then it would fail to satisfy the parametricity condition.

The *existential quantifier* works in a dual fashion. If $T(X, Z)$ is a binary type operator then we have a unary type operator $\exists Z.\ T(X, Z)$ which represents data abstractions that hide a representation type $Z$. To define it, consider "data type implementation" pairs of the form $\langle Z, p\rangle$ where $Z \in \mathcal{S}$ and $p \in T(X, Z)$. If $\langle Z, p\rangle$ and $\langle Z', p'\rangle$ are two implementations and $S : Z \leftrightarrow Z'$ is a relation such that

$$p\ [T(\Delta_X, S)]\ p'$$

we say that $S$ is a *simulation* relation, and that the two implementations are *similar*. For example, the abstract state machines $M$ and $M'$ for counters, mentioned in Section 2, are similar (where the type operator $T(X, Z)$ is $Z \times \{val\colon Z \to Int,\ inc\colon Z \to Z\}$.) We write $\langle Z, p\rangle \sim \langle Z', p'\rangle$ to denote that two implementations are similar.

The similarity relation $\sim$ is reflexive and symmetric.[9] So, its transitive closure $\sim^*$ is an equivalence relation. Write the equivalence class of $\langle Z, p\rangle$ under $\sim^*$ as $\langle\!|Z, p|\!\rangle$. The type operator $\exists^1(T)$ (denoted informally by the type expression $\exists Z.\ T(X, Z)$) is defined as follows:

[9] The reflexivity of $\sim$ is witnessed by the identity simulation relation, and symmetry by the converse-relation construction. Similarity is not transitive, however. The composition of two simulation relations is not necessarily a simulation relation. Consider, for example, the type operator $T(X, Z) = (Z \to Z) \to X$. Further discussion of this issue may be found in [33].

• The set part maps a set $X$ to the set $\exists Z.\, T(X,Z)$ whose elements are equivalence classes of implementations under the equivalence relation $\sim^*$.

• The relation part maps a relation $R : X \leftrightarrow X'$ to the relation $\exists S.\, T(R,S) : \exists Z.\, T(X,Z) \leftrightarrow \exists Z.\, T(X',Z)$, which is the least relation such that

$$\langle\!|Z,p|\!\rangle\ \exists S.\, T(R,S)\ \langle\!|Z',p'|\!\rangle \Longleftarrow \exists S\colon Z \leftrightarrow Z'.\ p\ T(R,S)\ p'$$

In other words, $\langle\!|Z,p|\!\rangle$ and $\langle\!|Z',p'|\!\rangle$ are related iff there exist $\langle Z_0,p_0\rangle \sim^* \langle Z,p\rangle$ and $\langle Z_0',p_0'\rangle \sim^* \langle Z',p'\rangle$ such that:

$$\exists S\colon Z_0 \leftrightarrow Z_0'.\ p_0\ T(R,S)\ p_0'$$

The intuition here is that the representation type $Z$ of the implementation $\langle Z,p\rangle$ is hidden from the client programs, and the client programs give the same results if we replace the implementation by a *similar* implementation $\langle Z',p'\rangle$. Hence all similar implementations are behaviorally equivalent. Identifying such behaviorally equivalent implementations is the semantic essence of data abstraction.

To complete the definition of the existential quantifier, we must verify that $\exists^1(T)$ has the identity extension property. (For the other operations, the identity extension property is already known [62].) We show this in two steps.

• $\exists S.\, T(\Delta_X,S) \subseteq \Delta_{\exists Z.\, T(X,Z)}$

If $\langle\!|Z,p|\!\rangle\ [\exists S.\, T(\Delta_X,S)]\ \langle\!|Z',p'|\!\rangle$, we have implementations $\langle Z_0,p_0\rangle \sim^* \langle Z,p\rangle$ and $\langle Z_0',p_0'\rangle \sim^* \langle Z',p'\rangle$ such that

$$\exists S : Z_0 \leftrightarrow Z_0'.\ p_0\ [T(\Delta_X,S)]\ p_0'$$

The relation $S$ is a simulation. Hence, $\langle Z_0,p_0\rangle \sim^* \langle Z_0',p_0'\rangle$ and $\langle\!|Z,p|\!\rangle = \langle\!|Z',p'|\!\rangle$.

• $\Delta_{\exists Z.\, T(X,Z)} \subseteq \exists S.\, T(\Delta_X,S)$.

If $\langle\!|Z,p|\!\rangle = \langle\!|Z',p'|\!\rangle$ then the identity relation $\Delta_Z : Z \leftrightarrow Z$ serves as the required relation $S$.

The first step in the above proof shows that the identification of behaviorally equivalent implementations is a necessary condition for the identity extension property.

The basic reference for parametricity is Reynolds [62], while Plotkin and Abadi [55] define a logic for reasoning about parametricity. The notion of existential quantification is from [43], but its parametricity semantics discussed above seems new. The idea of simulation relations for implementations dates back to Milner [40] and appears in various sources including [9, 33, 26, 44, 63].

The types $\theta$ of $\mathrm{IA}^+$ are interpreted as type operators $[\![\theta]\!]$ in the above sense. For completeness, we indicate the relation parts of these type operators.

$$\begin{array}{rcl}
[\![\mathsf{exp}[\delta]]\!](R) & = & R \to \Delta_{[\![\delta]\!]} \\
[\![\mathsf{comm}]\!](R) & = & R \to R \\
[\![\mathsf{val}[\delta]]\!](R) & = & \Delta_{[\![\delta]\!]} \\
[\![\theta_1 \times \theta_2]\!](R) & = & [\![\theta_1]\!](R) \times [\![\theta_2]\!](R) \\
[\![\theta \to \beta]\!](R) & = & \forall S.\, [\![\theta]\!](R \times S) \to [\![\beta]\!](R \times S) \\
[\![\mathsf{cls}\,\theta]\!](R) & = & \exists S.\, [\![\theta]\!](R \times S) \times [R \to R \times S]
\end{array}$$

The type operators of Algol types have additional structure. Whenever $R : Q \leftrightarrow Q'$ is the graph of a bijection $Q \cong Q'$, $[\![\theta]\!](R)$ is a bijection $[\![\theta]\!](Q) \cong [\![\theta]\!](Q')$. Further, as explained in [51, Sec. 3.2], there are certain "expand" functions which allow us to map a value in a small state to a related value in large state. Whenever $Q' \cong Q \times X$ is an expansion of the state set $Q$, there is a function $expand_\theta$ of type:

$$expand_\theta[Q, Q'] : [\![\theta]\!](Q) \to [\![\theta]\!](Q')$$

(Mathematically, this means that the type operators $[\![\theta]\!]$ are functors from a certain category of state sets to the category of sets.) We use the abbreviated notation $v{\uparrow}_Q^{Q'}$ to denote $expand_\theta[Q, Q'](v)$. The expanded value $v{\uparrow}_Q^{Q'}$ has the same action in the state set $Q'$ as $v$ has in $Q$. For example, if $\theta = \mathsf{comm}$ and $a \in [\![\mathsf{comm}]\!](Q)$, the expansion of $a$ to $Q \times X$ is:

$$a{\uparrow}_Q^{Q\times X} = \lambda(q, x).\ (a(q), x)$$

The expanded command has the same action as $a$ in that it transforms the $Q$ component via $a$ and leaves the extra state component unchanged. The definition of expand functions for Algol types may be found in [51, Sec. 3.2]. For $\theta = \mathsf{cls}\,\theta'$, the expand function is defined by:

$$\begin{aligned}&\langle\!| Z, (m, i) |\!\rangle{\uparrow}_Q^{Q\times X} \ = \\ &\quad \langle\!| Z, (m{\uparrow}_{Q\times Z}^{Q\times X\times Z},\ \lambda(q, x).\ (q', x, z_0) \text{ where } (q', z_0) = i(q)) |\!\rangle\end{aligned}$$

*Term interpretation*

The interpretation of terms is as follows. A term $M$ of type $\theta$ with free identifiers $x_1{:}\,\theta_1, \ldots, x_n{:}\,\theta_n$ is a parametrically polymorphic function

$$[\![M]\!] : \forall Q.\ [\![\{x_1{:}\,\theta_1, \ldots, x_n{:}\,\theta_n\}]\!](Q) \to [\![\theta]\!](Q)$$

So, for each state set $Q$, the meaning of $M$ has a component $[\![M]\!]_Q$ that maps records of type $[\![\{x_i : \theta_i\}_i]\!](Q)$ — which we call "environments" — to values of type $[\![\theta]\!](Q)$. Moreover, these components satisfy:

- the parametricity condition: for all relations $R : Q \leftrightarrow Q'$,

$$\eta\ [\![\{x_i : \theta_i\}_i]\!](R)\ \eta' \Longrightarrow [\![M]\!]_Q(\eta)\ [\![\theta]\!](R)\ [\![M]\!]_{Q'}(\eta')$$

- the naturality condition: whenever $Q' \cong Q \times X$,

$$[\![M]\!]_{Q'}(\eta{\uparrow}_Q^{Q'}) \ = \ [\![M]\!]_Q(\eta){\uparrow}_Q^{Q'}$$

The semantics of Algol phrases is as in [51]. We specify the interpretation of class constructs:

$$\begin{aligned}&[\![\mathbf{class} : \ \theta\ \mathbf{local}\, C\ x\ \mathbf{init}\ A\ \mathbf{meth}\ M]\!]_Q(\eta) = \\ &\quad \langle\!| Z,\ ([\![M]\!]_{Q\times Z}(\eta'),\ [\![A]\!]_{Q\times Z}(\eta') \circ i') |\!\rangle \\ &\qquad \text{where } \langle\!| Z,\ (m', i') |\!\rangle = [\![C]\!]_Q(\eta) \text{ and } \eta' = \eta \uparrow_Q^{Q\times Z} [x \to m']\end{aligned}$$

$$\begin{aligned}&[\![\mathbf{new}\ C\ P]\!]_Q(\eta) = \\ &\quad fst \circ p_Z(m) \circ i \\ &\qquad \text{where } \langle\!| Z, (m, i) |\!\rangle = [\![C]\!]_Q(\eta) \text{ and } p = [\![P]\!]_Q(\eta)\end{aligned}$$

A class definition builds an abstract type. This involves giving the representation state set for the objects of the class, the operations for the method suite, and the initialization operation. The **new** operator "opens" the abstract type and instantiates the client procedure $P$ with the representation state set obtained from the abstract type. Thus it is that an "instance" is created. In the normal case where $P$ is an abstraction $\lambda x.\, N$, its meaning is a family $\{\lambda m{:}\, [\![\theta]\!](Q \times Z).\, [\![N]\!]_{Q\times Z}(\eta\uparrow_Q^{Q\times Z}[x \to m])\}_Z$. So, the body term $N$ will now use the expanded state set $Q \times Z$. Every time the class $C$ is instantiated, a new $Z$ component is added to the state set in this fashion. Thus, every "opening" of the abstract type gives rise to a new instance with its own state component that is independent of all other state components.

*Remark.* In comparing this operation with the object encoding proposed by Pierce, Turner and others [54, 12], we note that they treat *objects* as abstract types whereas we treat *classes* as abstract types. Our objects correspond to "opened abstract types" whose representation types are merged into the global state set. Sending a message to the object merely involves selecting a component of its method suite. This is in contrast to the Pierce-Turner encoding where sending a message involves opening the object and repacking the results again to form a new object. Such repeated opening-closing operations are not present in the use of objects in Algol-like languages.

The interpretation of subtyping is by coercions. For each derivable subtyping $\theta <: \theta'$, we assign a coercion function of type:

$$[\![\theta <: \theta']\!] : \forall Q.\, [\![\theta]\!](Q) \to [\![\theta']\!](Q)$$

This is used in interpreting the Subsumption type rule. The coercions for the basic subtypings are the evident ones. For derived subtypings, we follow the general scheme as in [42, Sec. 10.4.2]. In particular, the interpretation of the width subtyping of records is the forgetting-fields coercion.

Finally, consider the interpretation of constants. A constant $c$ of type $\theta$ must be interpreted as a parametrically polymorphic family $[\![c]\!] \in \forall Q.\, [\![\theta]\!](Q)$ subject to the naturality condition: $[\![c]\!]_{Q'} = [\![c]\!]_Q\uparrow_Q^{Q'}$. The naturality condition implies that the entire family $[\![c]\!]$ is uniquely determined by its component at the singleton state set $\mathbf{1}$ (because every state set $Q$ is an expansion of $\mathbf{1}$). Hence we only need to specify the interpretation at the singleton state set.

Here is the interpretation of the class constants:

$$
\begin{array}{l}
[\![\mathrm{Var}[\delta]]\!]_{\mathbf{1}} = \\
\quad \langle\!| [\![\delta]\!],\ (\{\mathrm{get} = \lambda d\colon [\![\delta]\!].\ d, \\
\qquad\quad \mathrm{put} = \{\lambda n\colon [\![\delta]\!].\ \lambda(d,x)\colon [\![\delta]\!] \times X.\ (n,x)\}_X\ \}, \\
\qquad\quad \lambda x\colon \mathbf{1}.\ \mathbf{init}_\delta) |\!\rangle \\
\\
([\![*]\!]_{\mathbf{1}})_Q(c_1, c_2) = \\
\quad \langle\!| Z_1 \times Z_2,\ ((m_1', m_2'), i_2' \circ i_1) |\!\rangle \\
\quad \text{where} \quad \langle\!| Z_1,\ (m_1, i_1) |\!\rangle = c_1 \\
\qquad\qquad\quad \langle\!| Z_2,\ (m_2, i_2) |\!\rangle = c_2 \\
\qquad\qquad\quad m_1' = m_1 \uparrow^{Q\times Z_1\times Z_2}_{Q\times Z_1} \\
\qquad\qquad\quad m_2' = m_2 \uparrow^{Q\times Z_1\times Z_2}_{Q\times Z_2} \\
\qquad\qquad\quad i_2' = \lambda(q, z_1).\ (q', z_1, z_2') \text{ where } (q', z_2') = i_2(q)
\end{array}
$$

The $\mathrm{Var}[\delta]$ class denotes a state set $[\![\delta]\!]$ with *get* and *put* operations on it. The $*$ operator combines two classes by joining their state sets. The method suites of the individual classes are expanded to operate on the combined state set and the respective initialization operations are sequenced.

The following results are based on a straightforward verification:

LEMMA 5.1. *The interpretation of terms satisfies the parametricity and naturality conditions.*

THEOREM 5.1. *All the equivalences of Sec. 3.0.7. hold in the model.*

*Semantics of specifications*

Specification logic can also be interpreted in this model to some extent. The modeling is not complete because there is no clear notion of "locations used" in a computation. Such a notion is involved in our intuitive idea of non-interference. However, the basic structure of specifications, including the specification of classes, finds a satisfactory interpretation.

To interpret a specification logic formula $\varphi$ in a typing context $\Gamma$, we use statements of this form

$$Q, \eta \models \varphi$$

where $Q$ is a state set and $\eta$ is an environment in $[\![\Gamma]\!](Q)$. We read this as "$\varphi$ holds in the state set $Q$ and environment $\eta$.". A formula $\varphi$ is said to be *valid* if it holds in all state sets and all environments.

The interpretation of Specification logic constructs is shown in Table 8. The interpretation of Hoare-triple formulas is standard. A non-interference formula $M \# N$ holds if there are independent parts $X$ and $Y$ of the current state set $Q$ such that the value of $M$ is the expansion of some value in the state set $X$ and the value of $N$ is the expansion of some value in $Y$ [47]. This captures the idea that $M$ and $N$ do not use any common storage locations. The interpretation of logical connectives follows the possible world semantics of intuitionistic logic. The

**TABLE 8**
**Interpretation of specifications**

$$
\begin{array}{rcl}
Q,\eta \models M =_\theta N & \iff & [\![M]\!]_Q\eta =_{[\![\theta]\!](Q)} [\![N]\!]_Q\eta \\
Q,\eta \models \{P\}A\{P'\} & \iff & \forall q,q' \in Q.\, [\![P]\!]_Q\eta q = \mathsf{true} \wedge [\![A]\!]_Q\eta q = q' \Longrightarrow \\
 & & \quad [\![P']\!]_Q\eta q' = \mathsf{true} \\
Q,\eta \models M \#_{\theta,\theta'} N & \iff & \exists X,Y,Z.\, Q \cong X \times Y \times Z \wedge \\
 & & \quad \exists a \in [\![\theta]\!](X).\, \exists b \in [\![\theta']\!](Y). \\
 & & \quad [\![M]\!]_Q(\eta) = a{\uparrow}_X^Q \wedge [\![N]\!]_Q(\eta) = b{\uparrow}_Y^Q \\
Q,\eta \models \varphi \Longrightarrow \varphi' & \iff & \forall Z.\, (Q \times Z,\ \eta{\uparrow}_Q^{Q\times Z} \models \varphi) \Longrightarrow (Q \times Z,\ \eta{\uparrow}_Q^{Q\times Z} \models \varphi') \\
Q,\eta \models \forall x{:}\theta.\, \varphi & \iff & \forall Z.\forall v \in [\![\theta]\!](Q \times Z).\, (Q \times Z,\ \eta{\uparrow}_Q^{Q\times Z}[x \to v]) \models \varphi \\
Q,\eta \models \exists x{:}\theta.\, \varphi & \iff & \exists v \in [\![\theta]\!](Q).\, (Q,\ \eta[x \to v] \models \varphi) \\
Q,\eta \models \mathbf{Inst}\ C\ x.\, \varphi & \iff & \exists \langle Z, \langle m, i\rangle\rangle \sim^* [\![C]\!]_Q\eta.(Q \times Z,\ \eta{\uparrow}_Q^{Q\times Z}[x \to m] \models \varphi)
\end{array}
$$

meaning of **Inst** $C\ x.\, \varphi$ is that $\varphi$ should hold for $x$, where $x$ is an instance of some implementation of the class $C$. It is not necessary to use the same implementation as that in the definition of $C$. Because all implementations of the class are behaviorally equivalent, any one of them can be used to show that the instances satisfy $\varphi$. We use this feature below in showing that classes meet equational specifications.

LEMMA 5.2. *The inference rules (7) and (8) for* **Inst***-specifications are sound.*

We were unable to validate the proof rules of the non-interference predicate in this model, and it is very likely that they do not hold. A more explicitly location-based approach, as in [46], seems necessary to validate these rules.

*Examples*

The meaning of the class *Counter*, from Sec. 2, can be calculated as follows:

$$
\begin{array}{l}
[\![\mathrm{Counter}]\!]_Q(\eta) = \\
\quad \langle\!| Int,\ (\{\mathrm{inc} = \lambda(q,n).\, (q, n+1), \mathrm{val} = \lambda(q,n).\, n\},\ \lambda q.\, (q,0))|\!\rangle
\end{array}
$$

The parameter $(q,n)$ appearing in the operations is a state of type $Q \times Int$. Here, $Q$ is the state set of the context in which the class definition appears and $Int$ is the representation state set of the class. Note that the context part of the state is ignored. This is because the class *Counter* is given by a closed term, and its meaning in a state set $Q$ is just the expansion of its meaning in the singleton state set $\mathbf{1}$.

Consider the following class as an alternative to *Counter*:

$$\mathrm{Counter}' = \mathbf{class}\text{: } \{\mathrm{inc}\text{: } \mathsf{comm},\ \mathrm{val}\text{: } \mathsf{exp}[\mathsf{int}]\}$$

**local** Var[int] st
**init** st.put 0
**meth**
$\{\text{inc} = (\text{st.put} := \text{st.get} - 1),$
$\text{val} = -\text{st.get}\ \}$

Its meaning can be similarly calculated as

$$[\![\text{Counter}']\!]_Q(\eta) = \langle\!| \mathit{Int},\ (\{\text{inc} = \lambda(q,n).\,(q, n-1), \text{val} = \lambda(q,n).\,-n\},\ \lambda q.\,(q,0))|\!\rangle$$

The two implementations are equivalent because there is a simulation relation $S\colon \mathit{Int} \leftrightarrow \mathit{Int}$ given by

$$n\ S\ m \iff n \geq 0 \wedge m = -n \tag{12}$$

which is preserved by the two implementations. Hence, the two abstractions (equivalence classes) are *equal*: $[\![\text{Counter}]\!] = [\![\text{Counter}']\!]$. Thus, the parametricity semantics gives an extremely useful proof principle for reasoning about equivalence of classes.

The implementation of queues shown in Table 7 does not directly satisfy the equational axioms given in Table 5. For example, the second axiom does not hold for the implementation. (The left hand side gives a state where $f = r = 1$ whereas the right hand side gives a state where $f = r = 0$.) However, according to the semantics of **Inst**-specifications, it is enough for some behaviorally equivalent implementation to satisfy the axioms. The class is then deemed to satisfy the **Inst**-specification. We illustrate this here by giving an abstract implementation of queues, using lists, which is behaviorally equivalent to the original one (Table 9). It is easy to verify that the abstract implementation satisfies the queue axioms. The equivalence of this abstract queue implementation with the original one can be shown using the simulation relation:

$$S : (\mathit{Int} \to \mathit{Int}) \times \mathit{Int} \times \mathit{Int} \leftrightarrow \mathit{List}\ \mathit{Int}$$
$$(m, i, j)\ S\ e \iff i \leq j \wedge m[i+1, \ldots, j] = e$$

The three components in the state of the *Queue* class are the state of the array $a$ (regarded as a function from integers to integers) and the values of the variables $f$ and $r$.

*Handling recursion*

The above semantics can be adapted to handle recursion using the strict function framework in [49]. We replace the various concepts in the set-theoretic semantics as follows:

| | |
|---|---|
| state sets | flat pointed cpo's |
| sets | pointed cpo's |
| relations | complete relations |

**TABLE 9**
**An abstract implementation of queues**

```
AbstractQueue =
  class queue
    local Var[list int] e
    init e := [ ]
    meth
        {init = (e := [ ]),
         ins = λx. (e := e@[x]),
         del = (if e = [ ] then skip else e := tl(e)),
         front = if e ≠ [ ] then hd(e) else 0 }
```

The interpretation of IA$^+$ types is

$$\begin{aligned}
[\![\mathsf{exp}[\delta]]\!](Q) &= Q \multimap [\![\delta]\!] \\
[\![\mathsf{comm}]\!](Q) &= Q \multimap Q \\
[\![\mathsf{val}[\delta]]\!](Q) &= [\![\delta]\!] \\
[\![\theta_1 \times \theta_2]\!](Q) &= [\![\theta_1]\!](Q) \times [\![\theta_2]\!](Q) \\
[\![\{x_i\colon \theta_i\}_i]\!](Q) &= \textstyle\prod_{x_i} [\![\theta_i]\!](Q) \\
[\![\theta_1 \to \theta_2]\!](Q) &= \forall Z.\, [\![\theta_1]\!](Q \otimes Z) \to [\![\theta_2]\!](Q \otimes Z) \\
[\![\mathsf{cls}\,\theta]\!](Q) &= \exists Z.\, [\![\theta]\!](Q \otimes Z) \times [Q \multimap Q \otimes Z]
\end{aligned}$$

where $\otimes$ denotes smash product, $\multimap$ denotes strict function space, and $\to$ denotes continuous function space. The quantifiers $\forall$ and $\exists$ are similar to the set-theoretic case. The ordering on $\forall Z.\, T(X, Z)$ is pointwise while that on $\exists Z.\, T(X, Z)$ is the least relation $\sqsubseteq$ such that

$$p \sqsubseteq_{T(X,Z)} p' \Longrightarrow \langle\!| Z, p |\!\rangle \sqsubseteq \langle\!| Z, p' |\!\rangle$$

The relation parts of the operators $\multimap$, $\to$ and $\forall$ are as in the set-theoretic case. For $\otimes$ and $\exists$, the relations $R \otimes S$ and $\exists S.\, T(R, S)$ are defined to be the least complete relations satisfying:

$$\begin{aligned}
&x\ R\ x' \wedge y\ S\ y' \Longrightarrow [(x, y)]\ R \otimes S\ [(x', y')] \\
&\exists S\colon Z \leftrightarrow Z'.\, p\ [T(R, S)]\ p' \Longrightarrow \langle\!| Z, p |\!\rangle\ [\exists S.\, T(R, S)]\ \langle\!| Z', p' |\!\rangle
\end{aligned}$$

Such least relations exist because complete relations are closed under arbitrary intersections.

### 5.2. Object-based semantics

The "object-based" semantics described in [58, 48] (see also [5]) treats objects as state machines and describes them purely by their observable behavior. The observable behavior is given in terms of event traces whose structure is determined by the type of the object. This is similar to how processes are described in the semantics of CSP or CCS. Since no internal states appear in the denotations, proving the equivalence of two classes reduces to proving the equality of their trace

sets. The object-based semantics, described in [58, 48], makes these ideas work for Idealized Algol. For simplicity, we consider a version of Idealized Algol with "Syntactic Control of Interference", where functions are only applied to arguments that they do not interfere with. This is the language treated in [58]. The reader is referred to this paper for all the background material for this section.

We start with the notion of a coherent space [24], which is a simple form of event structure [69]. A *coherent space* is a pair $A = (|A|, \frown_A)$ where $|A|$ is a (countable) set and $\frown_A$ is a reflexive-symmetric binary relation on $|A|$. The elements of $|A|$ are to be thought of as *events* for the objects of a particular type. The relation $\frown_A$, called the *coherence relation*, states whether two events can possibly be observed from the same object in the same state.

The *free object space* generated by $A$ is a coherent space $A^* = (|A|^*, \frown_{A^*})$ where $|A|^*$ is the set of (finite) sequences over $|A|$ ("traces") and $\frown_{A^*}$ is defined by

$$\begin{array}{l} \langle a_1, \ldots, a_n \rangle \frown_{A^*} \langle b_1, \ldots, b_m \rangle \iff \\ \quad \forall i = 1, \ldots, min(n, m). \\ \quad\quad \langle a_1, \ldots, a_{i-1} \rangle = \langle b_1, \ldots, b_{i-1} \rangle \Longrightarrow a_i \frown_A b_i \end{array}$$

This states that, after carrying out a sequence of events $\langle a_1, \ldots, a_{i-1} \rangle$, the two traces must have coherent events at position $i$. If $a_i = b_i$, then the same condition applies to position $i + 1$. But if $a_i \neq b_i$, then the two events lead to distinct states and, so, there is no coherence condition on future events.

An *element* of a coherent space $A$ is a pairwise coherent subset $x \subseteq |A|$. So, the elements of free object spaces denote trace sets for objects. Functions appropriate for these spaces are what are called *regular maps* $f : A^* \to B^*$, defined in [58]. It turns out that regular maps can be described more simply in terms of *linear maps* of type $A^* \to B$. We actually define "multiple-argument linear maps" because they are needed for the term interpretation. A *linear map* of the form $F : A_1^*, \ldots, A_k^* \to B$ is a relation $F \subseteq (|A_1|^* \times \ldots \times |A_k|^*) \times |B|$ such that, whenever $(\vec{s}, b), (\vec{s'}, b') \in F$, we have

$$(\forall i.\, s_i \frown_{A_i^*} s_i') \implies b \frown_B b' \wedge (b = b' \Longrightarrow \vec{s} = \vec{s'})$$

(We are using the notation $\vec{s} = (s_1, \ldots, s_k)$ for the members of $|A_1|^* \times \cdots \times |A_k|^*$.) Every such linear map denotes a multiple-argument regular map $F^* : A_1^*, \ldots, A_k^* \to B^*$ given by

$$F^* = \{(\vec{s_1} \cdots \vec{s_n}, \langle b_1, \ldots, b_n \rangle) \mid (\vec{s_1}, b_1), \ldots, (\vec{s_n}, b_n) \in F\}$$

Coherent spaces for the events of various Idealized Algol types are shown in Table 10. For each IA type $\theta$, there is a coherent space which we also denote (ambiguously) by $\theta$. The symbols $a, b, \ldots$ are used for denoting events, $s, s', \ldots$ for event traces, and $i$ and $l$ for the labels in disjoint unions of sets. The trace sets for objects of type $\theta$ are the elements of $\theta^*$. Since we have a state-free description of objects, there is no characteristic difference between objects and classes as in the parametricity semantics. The only difference is that a class can be used repeatedly to generate new instances. So, a trace of a class is a sequence of object traces, one for each instance generated. Therefore, we define

$$\mathsf{cls}\ \theta = \theta^*$$

| | | | |
|---|---|---|---|
| $\lvert\mathsf{exp}[\delta]\rvert$ | $=$ | $[\![\delta]\!]$ | $a \subset b \iff a = b$ |
| $\lvert\mathsf{comm}\rvert$ | $=$ | $\{*\}$ | $* \subset *$ |
| $\lvert A_1 \times A_2\rvert$ | $=$ | $\lvert A_1\rvert + \lvert A_2\rvert$ | $(i,a) \subset (i',a') \iff (i = i' \Longrightarrow a \subset_{A_i} a')$ |
| $\lvert\{l_i\colon A_i\}_i\rvert$ | $=$ | $\Sigma_{l_i} A_i$ | $(l,a) \subset (l',a') \iff (l = l' \Longrightarrow a \subset_{A_l} a')$ |
| $\lvert A \to B\rvert$ | $=$ | $\lvert A^*\rvert \times \lvert B\rvert$ | $(s,b) \subset (s',b') \iff (s \subset_{A^*} s' \Longrightarrow b \subset_B b' \wedge (b = b' \Longrightarrow s = s'))$ |

**TABLE 10**
**Coherent spaces of events for IA types**

The meaning of a term $x_1\colon\theta_1, \ldots, x_n\colon\theta_n \rhd M : \theta$ is a multiple-argument linear map

$$[\![M]\!] : {\theta_1}^*, \ldots, {\theta_n}^* \to \theta$$

We regard a vector of traces $\vec{s} \in |\theta_1|^* \times \ldots \times |\theta_n|^*$ as a record $\eta \in \Pi_{x_i}|\theta_i|^*$. So, the linear map $[\![M]\!]$ is a set of pairs $(\eta, a)$, each of which indicates that, to produce an event $a$ for the result, the term $M$ carries out the event traces $\eta(x_i)$ on the objects for the free identifiers.

The interpretation of interference-controlled Algol terms is as in [58]. The interpretation of class terms is as follows (defined with reference to their typing rules):

$$\begin{aligned}
&[\![\textbf{class} : \theta\ \textbf{local}\, C\ x\ \textbf{init}\, A\ \textbf{meth}\, M]\!] = \\
&\quad \{(\eta_1 \cdot \eta_2 \cdot \eta_3, t) \mid \exists s_0, s_1 \in \tau^*. \\
&\qquad (\eta_1, s_0 \cdot s_1) \in [\![C]\!], \\
&\qquad (\eta_2[x \to s_0], *) \in [\![A]\!], \\
&\qquad (\eta_3[x \to s_1], t) \in [\![M]\!]^*\}
\end{aligned}$$

$$\begin{aligned}
&[\![\textbf{new}\, C\ P]\!] = \\
&\quad \{(\eta_1 \oplus \eta_2, *) \mid \exists s \in \theta^*. \\
&\qquad (\eta_1, s) \in [\![C]\!], \\
&\qquad (\eta_2, (s, *)) \in [\![P]\!]\}
\end{aligned}$$

The notation used in these definitions is as follows. The concatenation of traces as well as the pointwise concatenation of records of traces is denoted by "$\cdot$", e.g., $s_0 \cdot s_1$ and $\eta_1 \cdot \eta_2$. If $\eta_1$ and $\eta_2$ are records with disjoint sets of labels then $\eta_1 \oplus \eta_2$ denotes their join. This occurs in the interpretation of **new** $C\,P$ because we are considering a language with Syntactic Control of Interference where $C$ and $P$ cannot share free identifiers.

The meaning of the **class** term says that the trace set of $C$ must have a trace $s_0 \cdot s_1$ where $s_0$ represents the effect of the initialization command $A$. If the methods term $M$ maps the trace $s_1 \in |\tau|^*$ to a trace $t \in |\theta|^*$, then $t$ is a possible trace for the new class. The meaning of **new** $C\,P$ finds a trace $s$ supported by $C$ such that $P$ is ready to accept an object with this trace. Of course, $C$ supports many traces. But, $P$ will use at most one of these traces.

A primary advantage of the object-based semantics is that, by finessing the state representation in denotations, it makes it easier to reason about equality. Recall

that, to show that a class implementation meets an equational specification, we have had to find an equivalent implementation where the equational axioms actually hold. This is because implementations often have distinct states that are observationally equivalent and equality verification has to take this into account. However, since the object-based semantics is a state-free description, equational axioms can be verified directly. For example, the equation $\mathrm{x.inc}; \mathrm{g}(\mathrm{x.val}) = \mathrm{g}(\mathrm{x.val}+1); \mathrm{x.inc}$ of the *Counter* class is verified by noting that

$$s \cdot \langle(\mathrm{inc}, *), (\mathrm{val}, k+1)\rangle \in \textsc{cnt} \iff s \cdot \langle(\mathrm{val}, k), (\mathrm{inc}, *)\rangle \in \textsc{cnt}$$

where CNT is the trace set of the counter objects, defined in Sec. 2. Similarly, the equational axioms of queues can be verified for the *Queue* class by calculating its trace set and testing for particular sequences of events.

## 6. MODULARITY ISSUES

In this section, we briefly touch upon the higher-level modularity issues relevant to object-oriented programming. Further work is needed in understanding these issues.

### 6.1. Types and classes

In most object-oriented languages, the notion of types and classes is fused into one. Such an arrangement is not feasible in $\mathrm{IA}^+$ because classes are first-class values and their equality is not decidable. For example, the classes (*Array c n*) and (*Array c n′*) are equal only if $n$ and $n'$ are equal. Such comparisons are neither feasible nor desirable. However, a tighter integration of classes with types can be achieved using *opaque subtypes* as in Modula-3, also called "partially abstract" types [14]. For example, the counter class may be defined as:

**newtype** counter <: {inc: comm, val: exp[int]}
**reveal** counter = {inc: comm, val: exp[int]}
**in**
    Counter = **class**: counter **local** ...
**end**

A client program only knows that *counter* is some subtype of the corresponding signature type and that *Counter* is of type cls *counter*. Thus, it can create instances of *Counter* and manipulate them using the visible interface of *counter*. The definition of *Counter*, on the other hand, is inside the abstraction boundary of the abstract type *counter*, and regards *counter* as being *equal* to the signature type. (This is needed to type check the definition of the class.) A similar use of partially abstract types is made in [53] for modeling friend functions.

By associating a partially abstract type with each class in this fashion, we obtain types that correspond to classes. However, this set-up is more flexible than simply treating classes as types. For instance, we can define two behaviorally equivalent classes with the same associated partially abstract type. Their instances will be regarded as substitutable for each other. Moreover, there are no issues of undecidable equality with partially abstract types.

To ensure that all classes that have an associated partially abstract type implement common behavior, we can specify requirements for partially abstract types. For example, the specification:

$$\begin{array}{l}\forall \text{x: counter} \\ \quad \forall \text{k: } \mathsf{val}[\mathsf{int}]. \\ \quad\quad \{\text{x.val} = \text{k}\}\ \ \text{x.inc}\ \{\text{x.val} = \text{k} + 1\}\end{array}$$

states that all values of type *counter* must have *inc* and *val* methods with the counter behavior. All **reveal** blocks of the type *counter* get a proof obligation to demonstrate that their use of the type *counter* satisfies the specification.

Other applications of partially abstract types for controlling visibility of methods may be found in [20].

### 6.2. Dynamic Objects

Typical languages of the Algol family provide dynamic storage via Hoare's [27] concept of "references" (pointers). An object created in dynamic storage (or heap storage) is accessed through a reference, which is then treated as a data value and becomes storable in variables. Some of the modern languages, like Modula-3 and Java, treat references implicitly (assuming that every object is automatically a reference). But it seems preferable to make references explicit because the reasoning principles for them are much harder and not yet well-understood.

To provide dynamic storage in $\mathrm{IA}^+$, we stipulate that, for every type $\theta$, we have a data type $\mathsf{ref}\ \theta$. The operations for references are roughly as follows:

$$\frac{\Gamma \vdash C : \mathsf{cls}\,\theta}{\Gamma \vdash \mathbf{newref}\ C : (\mathsf{val}[\mathsf{ref}\ \theta] \to \mathsf{comm}) \to \mathsf{comm}}$$

$$\frac{\Gamma \vdash M : \mathsf{val}[\mathsf{ref}\ \theta]}{\Gamma \vdash M\!\uparrow : \theta}$$

The rule for **newref** is not sound in general. Since references can be stored in variables and exported out of their scope, they should not refer to any local variables that obey the stack discipline. If and when the local variables are deallocated, these references would become "dangling references." Or, put another way, the stack discipline of local variables breaks down. A correct type rule for **newref** is given in the Appendix.

Our knowledge of semantics for dynamic storage is rather incomplete. While some semantic models exist [64, 65], it is not yet clear how to integrate them with the reasoning principles presented here.

## 7. CONCLUSION

Reynolds's Idealized Algol is a quintessential foundational system for Algol-like languages. By extending it with objects and classes, we hope to provide a similar foundation for object-oriented languages based on Algol. In this paper, we have shown that the standard theory of Algol, including its equational calculus, specification logic and the major semantic models, extends to the object-oriented setting. In fact, much of this has been already implicit in the Algol theory but perhaps in a form accessible only to specialists.

Among the issues we leave open for future work are a more thorough study of inheritance models, reasoning principles for references, and investigation of call-by-value Algol-like languages.

## APPENDIX: REFLECTIVE TYPE CLASSES

In stating the equational properties of Sec. 3.0.7., we assumed that classes were given by closed terms. This is too severe an assumption. Typical class definitions are not closed terms, but they have free identifiers for constant values, class names etc. One still expects such classes to satisfy the properties mentioned in 3.0.7. because they do not have global side effects. A reasonable relaxation is to allow free identifiers but only if it is known that they refer to other quantities that are free of global side effects as well. This kind of restriction is also useful in other contexts, e.g., for defining "function procedures" that read global variables but do not modify them [65, 67].

The use of dynamic storage involves a similar restriction. A class used to instantiate a dynamic storage object should not have any references to local store. We define a general notion that is useful for formalizing such restrictions.

DEFINITION A.1. A *reflective type class* is a set of type terms $T$ such that

1. $\tau_1, \tau_2 \in T \Longrightarrow \tau_1 \times \tau_2 \in T$
2. $\tau \in T \Longrightarrow \theta \to \tau \in T$
3. $\tau_1, \ldots, \tau_n \in T \Longrightarrow \{x_1\colon \tau_1, \ldots, x_n\colon \tau_n\} \in T$

The terminology is motivated by the fact that these classes can be interpreted in reflective subcategories of the semantic category [57].

We define several reflective type classes based on the following intuitions. *Constant types* involve values that are state-independent; they neither read nor write storage locations. (Such values have been called by various qualifications such as "applicative" [65], "pure" [45], and "chaste" [66]). Values of *passive types* read storage locations, but do not write to them (one of the senses of "const" in C++). Values of *dynamic types* access only dynamic storage via references.

We add three new type constructors Const, Pas and Dyn which identify the values with these properties:

$$\theta \ ::= \ \ldots \mid \mathsf{Const}\,\theta \mid \mathsf{Pas}\,\theta \mid \mathsf{Dyn}\,\theta$$

A value of type $\mathsf{Const}\,\theta$ is a $\theta$-typed value that has been built using only constant-typed information from the outside. So it can be regarded as a constant value.

We define the following classes as the least reflective classes satisfying the respective conditions:

1. *Constant types* include $\mathsf{val}[\delta]$ and $\mathsf{Const}\,\theta$ types.
2. *State-dependent types* include $\mathsf{exp}[\delta]$ and comm, and are closed under Const, Pas and Dyn type constructors.
3. *Passive types* include $\mathsf{val}[\delta]$, $\mathsf{exp}[\delta]$, $\mathsf{Const}\,\theta$ and $\mathsf{Pas}\,\theta$ types.
4. *Dynamic types* include $\mathsf{val}[\delta]$, $\mathsf{Const}\,\theta$ and $\mathsf{Dyn}\,\theta$ types.

DEFINITION A.2. If $\Gamma \vdash M : \theta'$, a free identifier $x{:}\,\theta$ in $\Gamma$ is said to be *$T$-used* in $M$ if every free occurrence of $x$ is in a subterm of $M$ with a $T$-type. (In particular, we say "constantly used", "passively used", and "dynamically used" for the three kinds of usages.)

The introduction rules for $\mathsf{Const}$, $\mathsf{Pas}$, and $\mathsf{Dyn}$ are as follows:

$$\frac{\Gamma \vdash M : \theta}{\Gamma \vdash M : \mathsf{Const}\,\theta} \quad \text{if } \Gamma \text{ is constantly-used in } M \text{ and there are no occurrences of } \uparrow.$$

$$\frac{\Gamma \vdash M : \theta}{\Gamma \vdash M : \mathsf{Pas}\,\theta} \quad \text{if } \Gamma \text{ and } \uparrow \text{ are passively used in } M.$$

$$\frac{\Gamma \vdash M : \theta}{\Gamma \vdash M : \mathsf{Dyn}\,\theta} \quad \text{if } \Gamma \text{ is dynamically used in } M.$$

The dereference operator ($\uparrow$) is treated as if it were an identifier; $\Gamma$ is $T$-used means that every identifier in $\Gamma$ is $T$-used. For the elimination of these type constructors, we use the subtypings (for all types $\theta$):

$$\begin{array}{l} \mathsf{Const}\,\theta \;<:\; \mathsf{Pas}\,\theta \;<:\; \theta \\ \mathsf{Const}\,\theta \;<:\; \mathsf{Dyn}\,\theta \;<:\; \theta \end{array}$$

Note that any closed term can be given a type of the form $\mathsf{Const}\,\theta$. For example, the counter class of Section 3 has the type $\mathsf{Const}\,(\mathsf{cls}\ \mathit{counter})$.

*Application to class definitions.* The type rule for classes is now modified as follows:

$$\frac{\Gamma \rhd C : \mathsf{cls}\,\tau \quad \Gamma,\, x{:}\,\tau \rhd M : \theta \quad \Gamma,\, x{:}\,\tau \rhd A : \mathsf{comm}}{\Gamma \rhd (\mathbf{class} :\ \theta\ \mathbf{local}\, C\ x\ \mathbf{init}\, A\ \mathbf{meth}\, M) : \mathsf{cls}\,\theta}$$

(if $\Gamma$ is passively used in $A$)

This allows the free identifiers $\Gamma$ to be used in $A$, but in a read-only fashion. The parametricity interpretation of $\mathsf{cls}$-type must be modified to $[\![\mathsf{cls}\theta]\!](Q) = \exists Z.\, [\![\theta]\!](Q \times Z) \times [Q \to Z]$. The rest of the theory remains the same, except that the equation (4) becomes conditional on non-interference:

$$c \# a \implies \mathbf{new}\, c\, \lambda x.\, a; g(x) \;=\; a; \mathbf{new}\, c\, g$$

*Application to references.* We use the following rule for creating references:

$$\frac{\Gamma \rhd C : \mathsf{Dyn}\,(\mathsf{cls}\,\theta)}{\Gamma \rhd \mathbf{newref}\, C : (\mathsf{val}[\mathsf{ref}\,\theta] \to \mathsf{comm}) \to \mathsf{comm}}$$

The rule ensures that the class instantiated in the dynamic store does not use any locations from the local store, so the instance will not use them either. This avoids the "dangling reference" problem.

## ACKNOWLEDGMENT

It is a pleasure to acknowledge Peter O'Hearn's initial encouragement in the development of this work as well as his continued feedback. John Reynolds, Bob Tennent, Hongseok Yang and the anonymous referees of FOOL 5 provided valuable observations that led to improvements in the presentation. This research was supported by the NSF grant CCR-96-33737.

## REFERENCES

1. M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1996.
2. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
3. M. Abadi and R. M. Leino. A logic of object-oriented programs. In *TAPSOFT '97 and CAAP/FASE*, volume 1214 of *LNCS*, pages 682–696. Springer-Verlag, 1997.
4. H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.
5. S. Abramsky and G. McCusker. Linearity, sharing and state. In *Algol-like Languages* [52], chapter 20.
6. P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 60–90. Springer-Verlag, 1990.
7. D. S. Andersen, L. H. Pedersen, H. Hüttel, and J. Kleist. Objects, types and modal logics. In *FOOL 4*, http://www.cs.indiana.edu/hyplan/pierce/fool/, 1997. Electronic proceedings.
8. V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *European Conf. on Object-oriented Programming*, 1998.
9. P. Borba and J. Goguen. Refinement of concurrent object-oriented programs. In S. Goldsack and S. Kent, editors, *Formal Methods and Object Technology*, chapter 11. Springer-Verlag, 1996.
10. 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.
11. K. B. Bruce. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP'95*, volume 952 of *LNCS*, pages 27–51. Springer-Verlag, 1995.
12. K. C. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, volume 1281 of *LNCS*, pages 415–438. Springer-Verlag, Berlin, 1997.
13. L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *LNCS*, pages 51–67. Springer-Verlag, 1984.
14. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1986.
15. W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Dep. of Computer Science, Brown Univ., May 1989. (Tech. Report CS-89-33).
16. W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *ACM Symp. on Princ. of Program. Lang.*, pages 125–135. ACM, 1990. (Reprinted as Chapter 14 of [25]).
17. O.-J. Dahl and K. Nygaard. An Algol-based simulation language. *Comm. ACM*, 9(9):671–678, September 1966.
18. P. DiBlasio and K. Fisher. A calculus for concurrent objects. In *CONCUR '96*, volume 1119 of *LNCS*, pages 655–670. Springer-Verlag, 1996.
19. J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. An interpretation of typed OOP in a language with state. *J. Lisp and Symbolic Comput.*, 8(4):357–397, 1995.
20. K. Fisher and J. C. Mitchell. On the relationship between classes, objects and data abstraction. *Theory and Practice of Object-oriented Systems*, 4(1):3–25, 1995.
21. K. Fisher and J. C. Mitchell. Classes = objects + data abstraction. Technical Report STAN-CS-TN-96-31, Stanford University, 1996.
22. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

23. D. R. Ghica. Semantics of dynamic variables in algol-like languages. Master's thesis, Queen's University, Kingston, Canada, Mar 1997. (available electronically from ftp://ftp.qucis.queensu.ca/pub/rdt).

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

25. C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

26. J. F. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. volume 213 of *LNCS*, pages 187–196. Springer-Verlag, 1986.

27. C. A. R. Hoare. Record handling. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 291–347. Academic Press, London, 1968.

28. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

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

30. M. Hoffman and B. C. Pierce. Positive subtyping. *Information and Computation*, 126:11–33, 1996.

31. S. Kamin. Inheritance in SMALLTALK–80: A denotational definition. In *ACM Symp. on Princ. of Program. Lang.*, January 1988.

32. S. N. Kamin and U. S. Reddy. Two semantic models of object-oriented languages. In Gunter and Mitchell [25], chapter 13.

33. Y. Kinoshita, P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. An axiomatic approach to binary logical relations with applications to data refinement. Manuscript, Queen Mary and Westfield, London, 1997.

34. K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In O. L. Madsen, editor, *ECOOP '97*, volume 615 of *LNCS*, pages 78–97. Springer-Verlag, 1992.

35. G. T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, pages 72–80, July 1991.

36. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

37. S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

38. I. A. Mason and C. L. Talcott. Axiomatizing operational equivalence in the presence of side effects. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 284–293. IEEE Computer Society Press, June 1989.

39. 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. (Reprinted as Chapter 7 of [52]).

40. 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.

41. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

42. J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1997.

43. J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

44. C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1992.

45. 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.

46. P. W. O'Hearn. A model for syntactic control of interference. *Math. Struct. Comput. Sci.*, 3:435–465, 1993.

47. P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In Brookes et al. [10]. (Reprinted as Chapter 18 of [52]).

48. P. W. O'Hearn and U. S. Reddy. Objects, interference and Yoneda embedding. In Brookes et al. [10].

49. P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. Electronic manuscript, Queen Mary and Westfield, London, April 1997. URL http://www.qmw.ac.uk/ ohearn.

50. 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.

51. P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995. (Reprinted as Chapter 16 of [52]).

52. P. W. O'Hearn and R. D. Tennent. *Algol-like Languages (Two volumes)*. Birkhäuser, Boston, 1997.

53. B. Pierce and D. N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, 1993.

54. Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

55. G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications - TLCA '93*, LNCS, pages 361–375. Springer-Verlag, 1993.

56. U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *ACM Symp. on LISP and Functional Program.*, pages 289–297. ACM, July 1988.

57. 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.

58. U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *J. Lisp and Symbolic Computation*, 9:7–76, 1996. (Reprinted as Chapter 19 of [52]).

59. J. C. Reynolds. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.*, pages 39–46. ACM, 1978. (Reprinted as Chapter 10 of [52]).

60. 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. (Reprinted as Chapter 3 of [52]).

61. 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. (Reprinted as Chapter 6 of [52]).

62. J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.

63. O. Schoett. Behavioral correctness of data representations. *Science of Computer Programming*, 14:43–57, 1990.

64. Ian Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1):77–107, February 1996.

65. V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In *Algol-like Languages* [52], chapter 9, pages 235–272.

66. R. D. Tennent. Semantical analysis of specification logic. *Inf. Comput.*, 85(2):135–162, 1990. (Reprinted as Chapter 13 of [52]).

67. R. D. Tennent. Denotational semantics. In S. Abramsky, D. M. Gabbay, and T. S. E Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 169–322. Oxford University Press, 1994.

68. G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 325–392. Springer-Verlag, 1987.

69. G. Winskel. An introduction to event structures. In 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*, pages 364–397. Springer-Verlag, 1989.