Objects and Classes in Algol-like Languages

Uday S. Reddy

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

Abstract

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. An important idea that comes to light is that classes are 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.

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 claim either direct or indirect descent from Algol. 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.

The formal framework we adopt is the technical notion of "Algol-like languages" defined by Reynolds [51]. 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 practical programming languages. This combination gives us an ideal setting for studying various programming language phenomena of relevance to practical languages like C++, Modula-3, Java etc.

Reynolds also argued [50, Appendix] that object-oriented programming concepts are implicit in his Idealized Algol. The essential idea is that classes correspond 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 [46, 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 comes to light is that classes are *abstract data types* whose theory corresponds to that of existential types [35]. In a sense, IA^+ is to Idealized Algol what SOL is to polymorphic lambda calculus. However, while SOL can be faithfully encoded in polymorphic lambda calculus [45], IA^+ is more constrained than Idealized Algol. The corresponding encoding does not preserve equivalences. Thus, IA^+ is a proper extension.

Related work A number of papers [19, 1, 11, 18] discuss object-oriented type systems for languages with side effects. It is not clear what contribution these type systems make to reasoning principles for programs. A related direction is that of "object encodings." Pierce and Turner [44] study the encoding of objects as abstract types, which bears some similarity to the parametricity semantics in this paper. More recent work along this line is [12]. Fisher and Mitchell [20] also relate classes to data abstraction. This work assumes a functional setting for objects, but some of the ideas deal with "state." Work on specification of stateful objects includes [5, 28, 29, 30] in addressing subtyping issues and [3, 6] in addressing self-reference issues. The major developments in the research on Algol-like languages are collected in [43]. Tennent [58] gives a gentle introduction to the concepts as of 1994.

2 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
- $\mathsf{exp}[\delta]$, the type of state-dependent *expressions* giving δ-typed values,

- $\mathsf{val}[\delta]$, the type of phrases that directly denote δ-typed *values* (without any state-dependence).

Here, δ ranges over a collection of "data types" such as int(eger) and bool(ean) whose values are storable in variables. The "types" like $\mathsf{exp}[\delta]$ and comm are called "phrase types" to distinguish them from data types. Values of arbitrary phrase types are not storable in variables.[1]

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 β ranges over base types ($\mathsf{exp}[\delta]$, comm and $\mathsf{val}[\delta]$). Except for $\mathsf{cls}\ \theta$ types, the remaining type structure is that of simply typed lambda calculus with record types and subtyping. See, for instance, Mitchell [34, Ch. 10] for details. The basic subtypings include

- $\mathsf{val}[\delta] <: \mathsf{exp}[\delta]$,
- $(\mathsf{val}[\delta] \to \theta) <: (\mathsf{exp}[\delta] \to \theta)$, for a collection of types θ called "state-dependent" types, and
- the standard record subtyping ("width" as well as "depth" subtyping).

Our interpretation of subtyping is by *coercions* [34, Sec. 10.4.2]. The parameter passing mechanism of IA^+ is call-by-name (as is usual with typed lambda calculus). The second coercion above makes available Algol's notion of call-by-value. An "expression" argument can be supplied where a "value" is needed.

The type $\mathsf{cls}\ \theta$ is the type of classes that describe the behavior of θ-typed objects. An "object" is an abstraction that encapsulates some internal state represented by "fields" and provides externally visible operations called "methods." A class defines the fields and methods for a collection of objects, which are then called its "instances." The distinction between classes and instances arises because objects are stateful. (If a class is stateless, then there is no observable difference between its instances and there would be little point in making the class-instance distinction.) Classes represent the abstract (or "mathematical") concept of a behavior whereas instances represent the concrete (or "physical") realizations of the behavior.

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

$$\begin{array}{l} \mathbf{class}\ \theta \\ \quad \mathbf{fields}\ C_1\, x_1; \ldots; C_n\, x_n \\ \quad \mathbf{methods}\ M \\ \quad \mathbf{init}\ A \end{array}$$

The various components of the description are as follows:

- θ 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 fields),
- $C_1, \ldots, C_n$ are terms denoting classes (of the respective fields),
- M is a term of type θ (defining the methods of the class), and
- A is a comm-typed term (for initializing the fields).

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.

It is noteworthy that we cannot define nontrivial classes without first having some primitive classes (needed for defining fields). We will assume a single primitive class for (mutable) variables via the constant:

$$\mathrm{Var}[\delta] : \mathsf{cls}\ \{\mathrm{get} : \mathsf{exp}[\delta],\ \mathrm{put} : \mathsf{val}[\delta] \to \mathsf{comm}\}$$

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.[2] We often use the abbreviation:

$$\mathsf{var}[\delta] \quad = \quad \{\mathrm{get} : \mathsf{exp}[\delta],\ \mathrm{put} : \mathsf{val}[\delta] \to \mathsf{comm}\}$$

for the signature type of variables. We assume the subtypings:

$$\begin{array}{l} \mathsf{var}[\delta] <: \mathsf{exp}[\delta] \\ \mathsf{var}[\delta] <: \mathsf{val}[\delta] \to \mathsf{comm} \end{array}$$

whose coercion interpretations are the corresponding field selections.

Note that the type $\mathsf{var}[\delta]$ is different from the class $\mathrm{Var}[\delta]$. Values of type $\mathsf{var}[\delta]$ need not be, in general, instances of $\mathrm{Var}[\delta]$. For instance, the following (trivial) class has instances of type $\mathsf{var}[\mathsf{int}]$:

$$\begin{array}{l} \mathrm{Trivial} = \mathbf{class} \\ \qquad\qquad \mathsf{var}[\mathsf{int}] \\ \qquad \mathbf{fields} \\ \qquad \mathbf{methods}\ \{\mathrm{get} = 0, \\ \qquad\qquad\qquad \mathrm{put} = \lambda \mathrm{k}.\ \mathsf{skip}\} \\ \qquad \mathbf{init}\ \mathsf{skip} \end{array}$$

Instances of this class always give 0 for the get message and do nothing in response to a put message. Yet they have the type $\mathsf{var}[\mathsf{int}]$. In essence, the type of an object merely gives its signature (the types of its methods), whereas its class defines its behavior. A tighter integration of classes and types would certainly be desirable. We return to this issue in Sec. 4.1.

As an example of a nontrivial class, consider the following class of counter objects:

$$\begin{array}{l} \mathrm{Counter} = \mathbf{class} \\ \qquad\qquad \{\mathrm{inc}\text{: } \mathsf{comm},\ \mathrm{val}\text{: } \mathsf{exp}[\mathsf{int}]\} \\ \qquad \mathbf{fields} \\ \qquad\qquad \mathrm{Var}[\mathsf{int}]\ \mathrm{cnt} \\ \qquad \mathbf{methods} \\ \qquad\qquad \{\mathrm{inc} = (\mathrm{cnt.put} := \mathrm{cnt.get} + 1), \\ \qquad\qquad\ \ \mathrm{val} = \mathrm{cnt.get}\ \} \\ \qquad \mathbf{init} \\ \qquad\qquad \mathrm{cnt.put}\ 0 \end{array}$$

A counter has a state variable for keeping a count; the *inc* method increments the count and the *val* method returns the count. (The definition of the *inc* method could have also been written as $\mathrm{cnt} := \mathrm{cnt} + 1$ using the subtypings of $\mathsf{var}[\delta]$. We use explicit coercions for clarity.)

[1] It is possible to postulate a data type of references (or pointers) $\mathsf{ref}\ \theta$, for every phrase type θ, 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. 4.3.

[2] We assume that all new variables come initialized to some specific initial value init_δ. It is also possible to use a modified primitive $\mathrm{Var}[\delta]\colon \mathsf{exp}[\delta] \to \mathsf{cls}\ \mathsf{var}[\delta]$ that allows explicit initialization via a parameter.

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

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

An instance of a class $C_1 * C_2$ is a pair consisting of an instance of C_1 and an instance of C_2. Other useful combinators abound. For instance, the following combinator is motivated by the work on "fudgets" [14]:

$$\mathtt{<>} : (\theta_1 \to \mathsf{cls}\ \theta_2) \times (\theta_2 \to \mathsf{cls}\ \theta_1) \to \mathsf{cls}\ (\theta_1 \times \theta_2)$$

An instance of $F_1 \mathtt{<>} F_2$ is a pair (a, b) where a is an instance of $F_1(b)$ and b an instance of $F_1(a)$. The two objects are thus interlinked at creation time using mutual recursion. Common data structures in programming languages such as arrays and records also give rise to class combinators. The array data structure can be regarded as a combinator of type:

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

so that (array C n) is equivalent to an n-fold product $C * \cdots * C$, viewed as a (partial) function from integers to C-objects. The record construction

$$\textbf{record}\ C_1\, x_1; \ldots; C_n\, x_n\ \textbf{end}$$

is essentially like $C_1 * \cdots * C_n$ except that its instances are records instead of tuples.

For creating instances of classes, we use the notation:

$$\textbf{new}\ C$$

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

new Counter λa. B

creates an instance of *Counter*, binds it to a and executes the command B. The scope of a extends as far to the right as possible, often delimited by parentheses or **begin-end** brackets.

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 θ 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 β-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 [51] and has been used in several other languages [37, 56].

2.1 The formal system

We assume a standard treatment for the typed lambda calculus aspects of IA$^+$. The type rules for cls types are shown in Fig. 1. Note that we have one rule for the introduction of cls types and one for elimination. We show a single field in a class term for simplicity. This is obviously not a limitation because the $*$ combinator of classes can be used to instantiate multiple classes. It is significant that the initialization command is restricted to acting on the field x. We do not allow it to alter arbitrary non-local objects. The **methods** term M, on the other hand, can act on non-local objects. This is useful, for instance, to obtain the effect of "static" fields in languages like C++ and Java. If a class term does not have any free identifiers, we call it a "constant class."

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

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

Figure 1: Type rules for cls types

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

Figure 2: Essential constants of IA$^+$

The restriction that the initialization command should have no free identifiers other than x is motivated by reasoning considerations. Programmers typically want to assume that the order of instance declarations is insignificant. If the initializations were to have global effects, the order would become significant. However, the restriction as stated in the rule is too stringent. One would want the initialization command to be able to at least *read* global variables. In Appendix A, we outline a more general type system based on the ideas of [50, 48] that allows read-only free identifiers.

The important constants of IA$^+$ are shown in Fig. 2. (The constants for expression and value types are omitted.) The constant 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 $\mathsf{val}[\delta']$.) The infix operator ":=" is a variant of **letval** defined by:

$$a := e \ \stackrel{\text{def}}{=}\ \textbf{letval}\ e\ a$$

For example, the command (cnt.put := cnt.get + 1) in the definition of the Counter class involves such sequencing. The **letval** operator is extended to higher types as follows:

$$\begin{aligned}&\textbf{letval}_{\delta,\theta_1\times\theta_2}\ e\ f\\ &\quad = (\textbf{letval}_{\delta,\theta_1}\ e\ (\mathit{fst} \circ f), \textbf{letval}_{\delta,\theta_2}\ e\ (\mathit{snd} \circ f))\\ &\textbf{letval}_{\delta,\theta_1\to\theta_2}\ e\ f\\ &\quad = \lambda x{:}\theta_1.\ \textbf{letval}_{\delta,\theta_2}\ e\ \lambda k.\ f\ k\ x\end{aligned}$$

Thus, all "state-dependent" types (as defined in Appendix A) have **letval** operators, and we have a coercion:

$$\lambda f.\,\lambda e.\,\mathbf{letval}\; e\; f : (\mathsf{val}[\delta] \to \theta) \to (\mathsf{exp}[\delta] \to \theta)$$

which serves to interpret the subtyping $(\mathsf{val}[\delta] \to \theta) <: (\mathsf{exp}[\delta] \to \theta)$.

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

$$\begin{array}{ll} (\beta) & \mathbf{new}\;(\mathbf{class}\;\theta\;\mathbf{fields}\;C\;x\;\mathbf{methods}\;M\;\mathbf{init}\;A) \\ & \qquad \equiv \lambda p.\,\mathbf{new}\;C\;\lambda x.\,A; p\;M \\ (\eta) & (\mathbf{class}\;\theta\;\mathbf{fields}\;C\;x\;\mathbf{methods}\;x\;\mathbf{init}\;\mathsf{skip}) \\ & \qquad \equiv C \\ (\gamma) & \mathbf{new}\;C_1\;\lambda x.\,\mathbf{new}\;C_2\;\lambda y.\,M \\ & \qquad \equiv \mathbf{new}\;C_2\;\lambda y.\,\mathbf{new}\;C_1\;\lambda x.\,M \end{array}$$

The (β) law specifies the effect of an Intro-Elim combination. The (η) law specifies the effect of an Elim-Intro combination where the "Elim" is the implicit elimination in field declarations. The (γ) law allows one to reorder **new** declarations. Note that it is important for initializations to be free from global effects for the (γ) law to hold.

The interaction of **new** declarations with various constants is axiomatized by the following equational axioms (for free identifiers $c : \mathsf{cls}\,\theta$, $a, b : \mathsf{comm}$, $f, g : \theta \to \mathsf{comm}$, $e : \mathsf{exp}[\delta]$, $h : \theta \to \mathsf{val}[\delta] \to \mathsf{comm}$ and $p : \mathsf{val}[\mathsf{bool}]$):[3]

$$\begin{array}{rcll} \mathbf{new}\;c\;\lambda x.\,\mathsf{skip} & = & \mathsf{skip} & (1) \\ \mathbf{new}\;c\;\lambda x.\,a; g(x) & = & a;\;\mathbf{new}\;c\;g & (2) \\ \mathbf{new}\;c\;\lambda x.\,g(x); b & = & \mathbf{new}\;c\;g;\;b & (3) \\ \left\{\begin{array}{l}\mathbf{new}\;c\;\lambda x. \\ \quad \mathbf{letval}\;e\;\lambda z.\,h\;x\;z\end{array}\right\} & = & \left\{\begin{array}{l}\mathbf{letval}\;e\;\lambda z. \\ \quad \mathbf{new}\;c\;\lambda x.\,h\;x\;z\end{array}\right\} & (4) \\ \mathbf{new}\;c\;\lambda x.\,\mathbf{if}\;p\;(f\;x)\;(g\;x) & = & \mathbf{if}\;p\;(\mathbf{new}\;c\;f)\;(\mathbf{new}\;c\;g) & (5) \end{array}$$

(In the presence of nontermination, the first equation must be weakened to an inequality $\mathbf{new}\;c\;\lambda x.\,\mathsf{skip} \sqsubseteq \mathsf{skip}$.) 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 its scope. Notice that we can derive from the second equation, by setting $g = \lambda x.\,\mathsf{skip}$, the famous equation:

$$\mathbf{new}\;c\;\lambda x.\,a \;=\; a \qquad (6)$$

which has been discussed in various papers on semantics of local variables [31, 32, 40]. 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 for all classes.

In [50, Appendix], Reynolds suggests encoding classes as their corresponding "new" operators. This involves the translation:

$$\begin{array}{l} \mathsf{cls}\,\theta \;\rightsquigarrow\; (\theta \to \mathsf{comm}) \to \mathsf{comm} \\ (\mathbf{class}\;\theta\;\mathbf{fields}\;C\;x\;\mathbf{methods}\;M\;\mathbf{init}\;A) \\ \qquad \rightsquigarrow\; \lambda p.\,\mathrm{new}\;C\;\lambda x.\,A; p(M) \\ \mathbf{new}\;C \;\rightsquigarrow\; C \end{array}$$

[3]Note that these axioms are equations of lambda calculus, not equational schemas. The symbols $c, a, g, \ldots$ are *free identifiers* which can never be substituted by terms that capture bound identifiers. For instance, in equation (2), a cannot be substituted by a term that has x occurring free.

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

2.2 Specifications

An ideal framework for specifying classes in IA$^+$ is the *specification logic* of Reynolds [52]. Specification logic is a theory within (typed) first-order intuitionistic logic (and, hence, its name is somewhat a misnomer). We use the intuitionistic connectives "&", "$\Longrightarrow$", "$\forall$" and "$\exists$". The types include those of Idealized Algol and an additional base type assert for assertions (state-dependent classical logic formulas). The atomic formulas of specification logic include:

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

Note that assertions form a "logic within logic." One can use classical reasoning for them even though the outer logic is intuitionistic. A non-interference formula $A \# B$ means intuitively that A and B do not access any common storage locations except in a read-only fashion. (A formal definition of the property uses a possible-world semantics [41].) We use a *symmetric* non-interference predicate (from [38]), which is somewhat easier to use than the original Reynolds's version. The proof rules for the non-interference predicate are the following:

1. $\ldots \& x_i \# y_j \& \ldots \Longrightarrow A \# B$ (where $\{x_i\}_i$ and $\{y_j\}_j$ are the free identifiers of A and B respectively).
2. $A \# B$ if both A and B are of "passive" types.
3. $A \# B$ if either A or B is of a "constant" type.

Passive types are those that give $\mathsf{exp}[\delta]$-typed values and constant types are those that give $\mathsf{val}[\delta]$-typed values. See Appendix A for further discussion. The effect of the non-interference predicate is best illustrated by the proof rule:

$$A \# B \Longrightarrow A; B = B; A$$

which states that two non-interfering commands can be freely reordered. The survey article of Tennent [58] has a detailed description of specification logic.

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

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

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

$$\begin{array}{l} \mathbf{Inst}\;\mathrm{Var}[\delta]\;\mathrm{x}. \\ \quad \forall \mathrm{p}: \mathsf{exp}[\delta] \to \mathsf{assert}.\;\mathrm{x} \# \mathrm{p} \Longrightarrow \\ \qquad \{\mathrm{p(k)}\}\;\mathrm{x.put}\;\mathrm{k}\;\{\mathrm{p(x.get)}\} \end{array}$$

$$
\begin{array}{l}
\text{Queue : } \mathsf{cls}\ \{\text{init: } \mathsf{comm},\ \text{ins: } \mathsf{val[int]} \to \mathsf{comm},\ \text{del: } \mathsf{comm},\ \text{front: } \mathsf{exp[int]}\ \} \\
\textbf{Inst}\ \text{Queue q.} \\
\quad \forall \text{x,y: } \mathsf{val[int]}.\ \forall \text{g: } \mathsf{exp[int]} \to \mathsf{comm}.\ \text{g} \# \text{q} \Longrightarrow \\
\qquad\quad \text{q.init; q.init} \;=\; \text{q.init} \\
\qquad \&\ \text{q.init; q.ins(x); q.del} \;=\; \text{q.init} \\
\qquad \&\ \text{q.ins(x); q.ins(y); q.del} \;=\; \text{q.ins(x); q.del; q.ins(y)} \\
\qquad \&\ \text{q.init; q.ins(x); g(q.front)} \;=\; \text{q.init; q.ins(x); g(x)} \\
\qquad \&\ \text{q.ins(x); q.ins(y); g(q.front)} \;=\; \text{q.ins(x); g(q.front); q.ins(y)}
\end{array}
$$

Figure 3: Equational specification of a queue class

$$
\begin{array}{l}
\textbf{Inst}\ \text{Queue q.} \\
\quad \exists \text{elems: } \mathsf{list\ val[int]} \to \mathsf{assert}. \\
\quad \forall \text{k: } \mathsf{val[int]}.\ \forall \text{s: } \mathsf{list\ val[int]}. \\
\qquad\quad \{\mathsf{true}\}\ \text{q.init}\ \{\text{elems([ ])}\} \\
\qquad \&\ \{\text{elems(s)}\}\ \text{q.ins(k)}\ \{\text{elems(s@[k])}\} \\
\qquad \&\ \{\text{elems(k::s)}\}\ \text{q.del}\ \{\text{elems(s)}\} \\
\qquad \&\ \{\mathsf{true}\}\ \mathsf{skip}\ \{\text{elems(k::s)} \Rightarrow \text{q.front} = \text{k}\}
\end{array}
$$

Figure 4: Hoare-triple specification of queues

$$
\begin{array}{l}
\text{Queue} = \\
\quad \textbf{class}\ \text{queue} \\
\qquad \textbf{fields}\ \ \text{(UnboundedArray Var[int]) a;} \\
\qquad\qquad\quad \text{Var[int] f, r} \\
\qquad \textbf{methods} \\
\qquad\quad \{\text{init} = (\text{f} := 0;\ \text{r} := 0), \\
\qquad\quad\ \ \text{ins} = \lambda\text{x}.\ (\text{r} := \text{r} + 1;\ \text{a(r.get)} := \text{x}), \\
\qquad\quad\ \ \text{del} = (\textbf{if}\ \text{f} < \text{r}\ \textbf{then}\ \text{f} := \text{f} + 1\ \textbf{else}\ \mathsf{skip}), \\
\qquad\quad\ \ \text{front} = \text{a(f.get + 1).get}\ \} \\
\qquad \textbf{init}\ (\text{f} := 0;\ \text{r} := 0)
\end{array}
$$

Figure 5: An implementation of queues

Thus, the Hoare logic axiom for assignment becomes an axiom of the variable class. One can also write equational specifications for classes. For example, consider the specification of counters by:

$$
\begin{array}{l}
\textbf{Inst}\ \text{Counter x.} \\
\quad \forall \text{g: } \mathsf{exp[int]} \to \mathsf{comm}.\ \text{x} \# \text{g} \Longrightarrow \\
\qquad \text{x.inc; g(x.val)} \;=\; \text{g(x.val + 1); x.inc}
\end{array}
$$

The quantified function identifier g plays the role of a "conversion" function, to convert expressions into commands. As a less trivial example, an equational specification of a Queue class is shown in Fig. 3. 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. As illustration, we show in Fig. 4, 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 list. (We are using an ML-like notation for lists.) Note that the logical facilities of specification logic allow us to specify the exitence of an abstraction function which would be implementation-dependent.

For example, Fig. 5 shows an implementation of the Queue class using "unbounded" arrays.[4] To show that it meets the Hoare-triple specification, we pick the predicate:

$$\text{elems(s)} \iff \text{f} \leq \text{r} \land \text{map a (f + 1 \ldots r)} = \text{s}$$

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. Note that the predicate incorporates both the "representation invariant" and the "representation function" in America's terminology [5]. In fact, all of America's theory for class specifications is implicit in specification logic.

[4] 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.

Specification logic is also able to express "history properties" recommended by Liskov and Wing [30]. For example, here is a formula that states that a counter's value can only increase over time:

$$
\begin{array}{l}
\textbf{Inst}\ \text{Counter x.} \\
\quad \forall \text{k: } \mathsf{val[int]}.\ \forall \text{g: counter} \to \mathsf{comm}.\ \text{x} \# \text{g} \Longrightarrow \\
\qquad \{\text{x.val} = \text{k}\}\ \text{g(x)}\ \{\text{x.val} \geq \text{k}\}
\end{array}
$$

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

$$
\frac{\textbf{Inst}\ C\,x.\,\phi(x) \qquad \begin{array}{c} \left[\begin{array}{c} \phi(x) \\ \{C \# A_i \Longrightarrow x \# A_i\}_i \end{array}\right] \\ \vdots \\ \psi(g\,x) \end{array}}{\psi(\textbf{new}\ C\ g)} \tag{7}
$$

where x does not occur free in any undischarged assumptions, the terms A_i and the formula $\psi(-)$. This states that, to prove a property ψ for $(\textbf{new}\ C\ g)$, we need to prove ψ for $(g\ x)$, where x is an arbitrary instance of C, assuming the specification $\phi(x)$ and the fact that x does not interference with anything unless C interferes with it. The terms A_i can be any terms whatever but, in a typical usage of the rule, they are the free identifiers of $\psi(g x)$. These non-interference assumptions arise from the fact that x is a "new" instance.

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

$$
\frac{\textbf{Inst}\ C\,z.\,\psi(z) \qquad \begin{array}{c} \left[\begin{array}{c} \psi(z) \\ \{C \# A_i \Longrightarrow z \# A_i\}_i \end{array}\right] \\ \vdots \\ \phi(M) \end{array}}{\textbf{Inst}\ (\textbf{class}\ \theta\ \textbf{fields}\ C\ z\ \textbf{init}\ A\ \textbf{methods}\ M)\ x.\,\phi(x)} \tag{8}
$$

where z does not occur free in any undischarged assumptions, the terms A_i and the formula $\phi(-)$.

Inst-specifications are not always adequate for capturing the entire behavior of class instances. Since they specify the behavior of instances in arbitrary states, they miss the specification of initial state and the final state transformations. Additional axioms involving **new**-terms are necessary to capture these aspects. For example, the Counter class satisfies the following "initialization" axiom:

$$\begin{aligned}&\mathbf{new}\ \text{Counter}\ \lambda \text{x}.\ \text{g(x.val); h(x)}\\&\quad = \mathbf{new}\ \text{Counter}\ \lambda \text{x}.\ \text{g(0); h(x)}\end{aligned}$$

which specifies that the initial value of a counter is 0. The "finalization" axiom:

$$\begin{aligned}&\mathbf{new}\ \text{Counter}\ \lambda \text{x}.\ \text{g(x); h(x.inc)}\\&\quad = \mathbf{new}\ \text{Counter}\ \lambda \text{x}.\ \text{g(x); h}(\mathsf{skip})\end{aligned}$$

states that any increment operations done just before deallocation are redundant.

3 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 [42], which highlights the data abstraction aspects of classes, and *object-based* semantics along the lines of [49], which highlights the class-instance relationship.

3.1 Parametricity semantics

As pointed by Reynolds [53], parametricity has to do fundamentally with data abstraction. Since classes incorporate data abstraction, one expects parametricity to play a role in their interpretation. We follow the presentation of [42, Sec. 2] in our discussion. In particular, we ignore recursion and curried functions. The later discussion in [42] in handling these features is immediately applicable.

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

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

(We normally write both T_{set} and T_{rel} as simply T, using the context to disambiguate the notation.) Similarly, n-ary type operators with n type variables can be defined.

The type operators for constant types, variable types, product and function space constructors are standard. For example, for the function space constructor, we have the relation part:

$$\begin{aligned}&f\ [T_1(R) \to T_2(R)]\ f' \iff\\&\quad (\forall x, x'.\, x\ T_1(R)\ x' \implies f(x)\ T_2(R)\ f'(x')\end{aligned}$$

The relation part for a constant type K is the identity relation, denoted Δ_K. We define quantified type operators for a universal quantifier $\forall$ and an existential quantifier $\exists$:

- The type operator $\forall Z.\, T(X, Z)$ represents *parametrically polymorphic functions* p with components $p_Z \in T(X, Z)$. Formally, its set part consists of all $\mathcal{S}$-indexed families $\{p_Z\}_{Z\in\mathcal{S}}$ such that, for all relations $S\colon Z \leftrightarrow Z'$, $p_Z\ \ T(\Delta_X, S)\ \ p_{Z'}$. Its relation part, which can be written as $\forall S.\, T(R, S)$ for any $R\colon X \leftrightarrow X'$, is defined by
$$\begin{aligned}&\{p_Z\}_{Z\in\mathcal{S}}\ \forall S.\, T(R,S)\ \{p'_Z\}_{Z\in\mathcal{S}}\\&\iff \forall Z, Z' \in \mathcal{S}.\, \forall S\colon Z \leftrightarrow Z'.\ p_Z\ T(R,S)\ p'_{Z'}\end{aligned}$$
- The operator $\exists Z.\, T(X, Z)$ represents *data abstractions* that implement an abstract type Z with operations of type $T(X, Z)$. To define it formally, consider "implementation" pairs of the form $\langle Z, p\rangle$ where $Z \in \mathcal{S}$ and $p \in T(X, Z)$. Two such implementations are said to be *similar*, $\langle Z, p\rangle \sim \langle Z', p'\rangle$, if there exists a relation $S\colon Z \leftrightarrow Z'$ such that
$$p\ T(\Delta_X, S)\ p'$$
(Any such relation S is termed a *simulation*.) The set part $\exists Z.\, T(X, Z)$ consists of equivalence classes of implementations under the equivalence relation $\sim^*$. Write the equivalence class of $\langle Z, p\rangle$ as $\langle\!| Z, p |\!\rangle$. The relation part $\exists S.\, T(R, S)$ for any relation $R\colon X \leftrightarrow X'$ is the least relation such that
$$\langle\!| Z, p |\!\rangle\ \exists S.\, T(R,S)\ \langle\!| Z', p' |\!\rangle \Longleftarrow \exists S\colon Z \leftrightarrow Z'.\ p\ T(R,S)\ p'$$

The basic reference for parametricity is Reynolds [53], while Plotkin and Abadi [45] define a logic for reasoning about parametricity. The notion of existential quantification is from [35], but the parametricity semantics is not mentioned there. The idea of simulation relations for abstract type implementations dates back to Milner [33] and appears in various sources including [9, 27, 25, 36, 54].

The types θ of IA$^+$ are interpreted as type operators $[\![\theta]\!]$ in the above sense. The parameters for the type operators are state sets. Typically they capture the states involved in the representation of objects. The relation parts of the operators specify how two values of type θ are related under change of representation. Here is the interpretation:

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

Note that the meaning of a class is a data abstraction. It involves a state set Z for the internal state of the instances, a component of type $[\![\theta]\!](Q \times Z)$ for the methods of the class and a component of type Z for the initial state. Two such implementations with internal state sets Z and Z' are similar (and, hence, equivalent) if, for some relation $S\colon Z \leftrightarrow Z'$, the initial states are related by S and their methods "preserve" S according to the relation $[\![\theta]\!](\Delta_Q \times S)$.

For example, consider the following class as an alternative to *Counter*:

$$\begin{aligned}\text{Counter2} = {}&\mathbf{class}\{\text{inc: }\mathsf{comm},\ \text{val: }\mathsf{exp}[\mathsf{int}]\}\\&\quad \mathbf{fields}\,\text{Var[int] st}\\&\quad \mathbf{methods}\\&\qquad \{\text{inc} = (\text{st.put} := \text{st.get} - 1),\\&\qquad\ \ \text{val} = -\text{st.get}\ \}\\&\quad \mathbf{init}\,\text{st.put } 0\end{aligned}$$

The meanings of *Counter* and *Counter2* can be calculated as follows:

$$\begin{aligned}&[\![\text{Counter}]\!]_Q(\eta) =\\&\quad \langle\!| \mathit{Int},\ (\{\text{inc} = \lambda(q,n).\,(q, n+1), \text{val} = \lambda(q,n).\,n\},\ 0) |\!\rangle\\&[\![\text{Counter2}]\!]_Q(\eta) =\\&\quad \langle\!| \mathit{Int},\ (\{\text{inc} = \lambda(q,n).\,(q, n-1), \text{val} = \lambda(q,n).\,-n\},\ 0) |\!\rangle\end{aligned}$$

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

$$n\ S\ m \iff n \geq 0 \wedge m = -n \qquad (9)$$

which is preserved by the two implementations. Hence, the two abstractions (equivalence classes) are *equal*: $[\![\text{Counter}]\!] = [\![\text{Counter2}]\!]$. Thus, the parametricity semantics gives an extremely useful proof principle for reasoning about equivalence of classes.

The interpretation of terms is as follows. A term M of type θ with free identifiers $x_1\colon\theta_1, \ldots, x_n\colon\theta_n$ is a parametric function

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

We write $[\![M]\!]_Q$ for the component of $[\![M]\!]$ at Q. The semantics of Algol phrases is as in [42]. An important point to recall from that paper, Sec. 3.2, is the fact that parametricity makes available certain "expand" functions:

$$\mathit{expand}_\theta[Q, Z] : [\![\theta]\!](Q) \to [\![\theta]\!](Q \times Z)$$

For every value $v \in [\![\theta]\!](Q)$, there is a unique expanded value in $[\![\theta]\!](Q \times Z)$ that acts the "same way" as v does. We use the abbreviated notation $v \uparrow_Q^{Q\times Z}$ to denote $\mathit{expand}_\theta[Q, Z](v)$. For example, if $\theta = \mathsf{comm}$ and $v \in [\![\mathsf{comm}]\!](Q)$, the expanded command

$$v \uparrow_Q^{Q\times Z} = \lambda(q, z).\, (v(q), z)$$

leaves the Z component unchanged. These expand functions play a crucial role in interpreting instance declarations and inheritance. They also have significance in interpreting constants. A "constant value" in $[\![\theta]\!](Q)$ is a value of the form $v \uparrow_1^Q$ obtained by expanding a value in the unit state set. So, we only need to specify the interpretation of a constant in the unit state set.

The semantics of class constructs is as follows:

$$\begin{array}{l}
[\![\mathbf{class}\ \theta\ \mathbf{fields}\ C\ x\ \mathbf{methods}\ M\ \mathbf{init}\ A]\!]_Q(\eta) = \\
\quad \langle\!| Z,\ ([\![M]\!]_{Q\times Z}(\eta'[x \to m']),\ [\![A]\!]_Z[x \mapsto m'](z')) |\!\rangle \\
\qquad \text{where } \langle\!| Z,\ (m', z') |\!\rangle = [\![C]\!]_Q(\eta) \text{ and } \eta' = \eta \uparrow_Q^{Q\times Z}
\end{array}$$

$$\begin{array}{l}
[\![\mathbf{new}\ C\ P]\!]_Q(\eta) = \\
\quad \lambda q.\, \mathit{fst}(p_Z(m)(q, z)) \\
\qquad \text{where } \langle\!| Z, (m, z) |\!\rangle = [\![C]\!]_Q(\eta) \text{ and } p = [\![P]\!]_Q(\eta)
\end{array}$$

A class definition builds an abstract type as illustrated with *Counter* above. The **new** operator "opens" the abstract type and passes to the client procedure P the representation and the method suite of the class. Thus, an "instance" is created. Note that, in the normal case where P is an abstraction $\lambda x.\, M$, its meaning is $\Lambda Z.\, \lambda m\colon [\![\theta]\!]_{Q\times Z}(\eta \uparrow_Q^{Q\times Z} [x \to m]$. So, the body term M 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 does not interfere with the others.

In comparing this operation with the object encoding proposed by Pierce, Turner and others [44, 12], we note that they treat *objects* as abstract types whereas we treat *classes* as abstract types. Thus, some of the bureaucratic opening-closing code that appears in their model is finessed here. Message send in our model is simply the field selection of a record. Nevertheless, the idea of abstract types appears in both the models, and the implications of this commonality should be explored further.

The class constants have the following interpretation:

$$\begin{array}{l}
[\![\mathrm{Var}[\delta]]\!]_1 = \\
\quad \langle\!| [\![\delta]\!],\ (\{\mathrm{get} = \lambda d.\, d, \\
\qquad\qquad \mathrm{put} = \Lambda X.\, \lambda n.\, \lambda(d, x).\, (n, x)\}, \\
\qquad\qquad \mathbf{init}_\delta) |\!\rangle
\end{array}$$

$$\begin{array}{l}
[\![*]\!]_1[Q](c_1, c_2) = \\
\quad \langle\!| Z_1 \times Z_2,\ ((m_1', m_2'), (z_1, z_2)) |\!\rangle \\
\quad \text{where} \quad \langle\!| Z_1,\ (m_1, z_1) |\!\rangle = c_1 \\
\qquad\qquad\ \ \langle\!| Z_2,\ (m_2, z_2) |\!\rangle = c_2 \\
\qquad\qquad\ \ m_1' = m_1 \uparrow_{Q\times Z_1}^{Q\times Z_1\times Z_2} \\
\qquad\qquad\ \ m_2' = m_2 \uparrow_{Q\times Z_2}^{Q\times Z_1\times Z_2}
\end{array}$$

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.

Theorem 1 *The parametricity model satisfies all the equivalences and axioms of Sec. 2.1.*

The plain parametricity semantics described above does not handle the equality relation in a general fashion. In implementing data abstractions, it is normal to allow the same abstract value to be represented by multiple concrete representations. In our context, this means that the equality relation for abstract states is, in general, not the same as the equality relation for concrete states. It corresponds to a partial equivalence relation (PER) for concrete states [24].

For example, in the Queue implementation of Fig. 5, an empty queue is represented by *any* state in which f and r are equal. The second axiom of the equational specification (Fig. 3) does not hold in this implementation. (The left hand side gives a state with $\mathrm{f} = \mathrm{r} = 1$ whereas the right hand side gives a state with $\mathrm{f} = \mathrm{r} = 0$.)

This can be remedied by modifying the parametricity semantics to a parametric PER semantics, where each type carries its own notion of equality.[5] More formally, A "type" in the new setting (called a PER-*type*) is a pair $X = \langle X, E_X \rangle$ where X is a set and E_X is a PER over X representing the notion of equality for X. All the above ideas can be modified to work with PER-types. (See Appendix B.)

The PER semantics influences reasoning about programs as follows. Suppose we obtain a package $\langle\langle Z, \Delta_Z \rangle, p\rangle \in \exists Z.\, T(X, Z)$ as the meaning of a class. If p preserves some PER E_Z in the sense that $p\ T(E_X, E_Z)\ p$ then we have

$$\langle\langle Z, \Delta_Z \rangle, p\rangle \sim \langle\langle Z, E_Z \rangle, p\rangle$$

Thus, we are at liberty to make up any PER E_Z that is preserved by p and use it as the equality relation for the representation.

For example, for the Queue class of Fig. 5, the state set Z consists of triples $\langle a, f, r\rangle$ where $a\colon \mathit{Int} \to \mathit{Int}$ and $f, r \in \mathit{Int}$. We pick the equivalence relation E_Z given by:

$$\begin{array}{l}
\langle a, f, r\rangle\ E_Z\ \langle a', f', r'\rangle \iff \\
\qquad f \le r \wedge f' \le r' \\
\qquad \wedge\ \mathrm{map}\ a\ (f+1 \ldots r) = \mathrm{map}\ a\ (f'+1 \ldots r')
\end{array}$$

to represent the intuition that only the portion of the array between $f+1$ and r contains meaningful values. In verifying the axioms of queues, we interpret $=_{\mathsf{comm}}$ as the PER for

[5] It does not seem possible to obtain the information of this semantics from the plain parametricity semantics because quantified type operators do not map PER's to PER's in general.

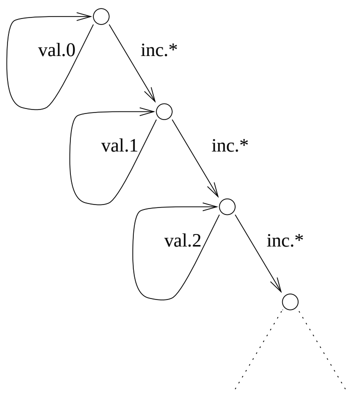

Figure 6: Trace set of a counter object

$[\![\mathsf{comm}]\!](Q \times Z)$, viz., $[E_Q \times E_Z \to E_Q \times E_Z]$. ($E_Q$ is some PER for Q respected by the other variables like g.) Here is the verification of the problematic second axiom. The two sides of the equation denote the respective state transformations:

$$\lambda(q, \langle a, f, r\rangle).\, (q,\, \langle a[1 \to x],\, 1,\, 1\rangle)$$
$$\lambda(q, \langle a, f, r\rangle).\, (q,\, \langle a,\, 0,\, 0\rangle)$$

It is clear that they are equivalent by the relation $[E_Q \times E_Z \to E_Q \times E_Z]$.

3.2 Object-based semantics

The object-based semantics [49, 39] (see also [4]) 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.

Before looking at formal definitions, we consider an example. Figure 6 depicts the trace set of a counter object in its initial state. 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 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 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:

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

The parameter of the CNT function is the label of the state. Note that these labels can be anything we make up, but it often 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:

$$\begin{aligned}&\mathrm{CNT2}(0) \text{ where}\\ &\quad \mathrm{CNT2}(m) = \{\epsilon\} \cup \{\mathrm{inc}.*\} \cdot \mathrm{CNT2}(m+1)\\ &\qquad\qquad\quad \cup \{\mathrm{val}.(-m)\} \cdot \mathrm{CNT2}(m)\end{aligned}$$

This description corresponds to the class *Counter2*. While it is obvious that the two trace sets are the same, a formal proof would use the simulation relation S defined in (9). We can show by fixed point induction that

$$n\ S\ m \quad\Longrightarrow\quad \mathrm{CNT}(n) = \mathrm{CNT2}(m)$$

and it follows that $\mathrm{CNT}(0) = \mathrm{CNT2}(0)$.

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 specification equations of classes can be directly verified in the trace sets. For example, the equation $\mathrm{x.inc}; \mathrm{g(x.val)} = \mathrm{g(x.val+1)}; \mathrm{x.inc}$ of the *Counter* class is verified by noting that

$$\langle \mathrm{inc}.*, \mathrm{val}.(k+1)\rangle \in \mathrm{CNT}(n) \iff \langle \mathrm{val}.k, \mathrm{inc}.*\rangle \in \mathrm{CNT}(n)$$

for all states n.

The object-based semantics, described in [47, 39], 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.

We start with the notion of a coherent space [22], which is a simple form of event structure [59]. 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 sequences over $|A|$ ("traces") and $\frown_{A^*}$ is defined by

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

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 object spaces denote trace sets for objects. Functions appropriate for object spaces are what are called *regular maps* $f : A^* \to B^*$. It turns out that they can be described more simply in terms of *linear maps* $F : A^* \to B$. We actually define "multiple-argument linear maps" because they are needed for semantics. A *linear map* $F : A_1^*, \ldots, A_k^* \to B$ 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

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

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 Figure 7. The trace sets for objects of type θ are elements of $[\![\theta]\!]^*$. Since we have a state-free description of objects, there is virtually no difference between

$$
\begin{array}{lcll}
|\mathsf{exp}[\delta]| & = & |\delta| & a \frown b \iff a = b \\
|\mathsf{comm}| & = & \{*\} & * \frown * \\
|A_1 \times A_2| & = & |A_1| + |A_2| & i.a \frown i'.a' \iff (i = i' \Longrightarrow a \frown_{A_i} a') \\
|\{l_i\colon A_i\}_i| & = & \Sigma_{l_i} A_i & l.a \frown l'.a' \iff (l = l' \Longrightarrow a \frown_{A_l} a') \\
|A^* \to B| & = & |A^*| \times |B| & (s, b) \frown (s', b') \iff (s \frown_{A^*} s' \Longrightarrow b \frown_B b' \wedge (b = b' \Longrightarrow s = s'))
\end{array}
$$

Figure 7: Coherent spaces of events for IA types

objects and classes. 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

$$[\![\mathsf{cls}\,\theta]\!] = [\![\theta]\!]^*$$

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 (η, 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 [49]. The interpretation of class terms is as follows:

$$
\begin{array}{l}
[\![\textbf{class}\ \theta\ \textbf{fields}\ C\ x\ \textbf{methods}\ M\ \textbf{init}\ A]\!] = \\
\quad \{(\eta_1 \cdot \eta_2 \cdot \eta_3, s) \mid \exists s_0, s_1 \in [\![\tau]\!]^*. \\
\qquad (\eta_1, s_0 s_1) \in [\![C]\!], \\
\qquad (\eta_2[x \to s_0], *) \in [\![A]\!], \\
\qquad (\eta_3[x \to s_1], s) \in [\![M]\!]^*\}
\end{array}
$$

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

The meaning of the **class** term says that the trace set of C must have a trace $s_0 s_1$ 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 $s \in |[\![\theta]\!]|^*$, then s 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.

Theorem 2 *The object-based model satisfies all the equivalences and axioms of Sec. 2.1, adapted to a version of IA$^+$ with Syntactic Control of Interference.*

4 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.

4.1 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 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 [21]. 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 ...

A client program only knows that *counter* is some subtype of the corresponding signature type and that *Counter* is of type cls *counter*. The class *Counter*, on the other hand, is inside the abstraction boundary of the abstract type *counter*, and regards it as being equal to the signature type.

We can specify requirements for partially abstract types. For example, the specification:

> ∀x: counter.
> ∀k: val[int]. ∀g: counter → comm. x # g ⟹
> {x.val = k} g(x) {x.val ≥ k}

states that *every* value of type counter — not just an instance of some class — is monotonically increasing. All **reveal** blocks of the type counter get a proof obligation to demonstrate that their use of the type counter satisfies the specification. For example, if we use **reveal** blocks to define classes Counter and Counter2, we have the job of showing that their instances are monotonically increasing. Note that such partially abstract types correspond to what America [5] calls "types."

4.2 Inheritance

Since IA$^+$ is a typed lambda calculus with records, most inheritance models in the literature can be adapted to it. For illustration, we show the recursive record model [13, 15, 46]. A class that uses self-reference is defined to be of type cls $(\theta \to \theta)$ instead of cls θ, so that the method suite is parameterized by "self." We have a combinator

> close: cls $(\theta \to \theta) \to$ cls θ
> close c = **class** θ **fields** c f **methods** fix f **init** skip

which converts a self-referential class c to a class whose instances are ordinary objects.

Let c be of type cls $(\theta \to \theta)$. To define a derived class of type cls$(\theta' \to \theta')$ where $\theta' = \theta \oplus \tau$ is a record type extension, we use a construction of the form:

> **class** $\theta' \to \theta'$
> **fields** c f; ...
> **methods** λself. (f self) **with**[τ] M
> **init** A

where **with** is a record-combination operator [16]. (The **with** operator is qualified by the type τ to indicate the record fields that get updated. This is needed for coherence of subtyping.)

As an example, suppose we define a variant of the Counter class that provides a *set* method in a "protected" fashion:

```
type protected_counter =
        counter ⊕ {set: val[int] → comm}
reveal counter = protected_counter
in
  Counter =
    class   counter → counter
      fields Var[int] cnt
      methods
          λself. {val = cnt.get,
                  set = cnt.put,
                  inc = (self.set := self.val + 1) }
      init cnt.put 0
```

We can then define a derived class that issues warnings whenever the counter reaches a specified limit:

```
reveal counter = protected_counter
in
  Warn_Counter lim =
    class   counter → counter
      fields Counter f
      methods
        λself. (f self) with[set:...]
            {set = λk. if k = lim then
                              print "Limit reached";
                        (f self).set k}
      init skip
```

Note that both (close Counter) and (close Warn_Counter) are of type cls counter. Their instances satisfy the specification of counter, including its history property. The *set* method does not cause a problem because it is inaccessible to clients.

The proof principle for self-referential classes is derived from fixed-point induction:

$$\phi(\bot) \wedge (\textbf{Inst}\ c\ f.\ \forall x.\ \phi(x) \Longrightarrow \phi(f(x))) \\ \Longrightarrow \textbf{Inst}\ (\text{close}\ c)\ x.\ \phi(x)$$

For example, both C = Counter and C = (Warn_Counter lim) satisfy:

```
Inst (close C) x.
  ∀k: val[int]. ∀p: exp[int] → assert.
      {p(k)} x.set k {p(x.val)}
   &  {p(x.val + 1)} x.inc {p(x.val)}
```

4.3 Dynamic Objects

Typical languages of Algol family provide dynamic storage via Hoare's [26] concept of "references" (pointers). An object created in dynamic 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, 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 IA^+, we stipulate that, for every type θ, we have a data type $\mathsf{ref}\ \theta$. The operations for references are roughly as follows:

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

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

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". A correct type rule for **newref** is given in Appendix A.

Our knowledge of semantics for dynamic storage is rather incomplete. While some semantic models exist [55, 56], it is not yet clear how to integrate them with the reasoning principles presented here.

5 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.

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

Appendix

A Reflective type classes

In the type rules of section 2.1, the initialization command of a class was restricted to only the local fields of the class. While this restriction leads to clean reasoning principles: the (γ) law and equations (2-6), it is too restrictive to be practical. For instance, a counter class parameterized by an initial value n does not type-check under this restriction because its **init** command has free occurrences of n. A reasonable relaxation of the restriction is to allow the initialization command to read storage locations, but not to write to them. 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 [58, 56].

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 3 *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. $\tau_1, \ldots, \tau_n \in T \Longrightarrow \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \in T$

The terminology is motivated by the fact that these classes can be interpreted in reflective subcategories of the semantic category [48].

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" [56], "pure" [37], and "chaste" [57]). Dually, *state-dependent types* involve values that necessarily depend on the state. Values of *passive types* only 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 even if they are of general types.

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

A value of type $\mathsf{Const}\theta$ is a θ-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 4 *If $\Gamma \vdash M : \theta'$, a free identifier $x : \theta$ in Γ 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 Const, Pas, and 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; Γ is T-used means that every identifier in Γ is T-used. For the elimination of these type constructors, we use the subtypings (for all types θ):

$$\begin{array}{ccccc} \mathsf{Const}\,\theta & <: & \mathsf{Pas}\,\theta & <: & \theta \\ \mathsf{Const}\,\theta & <: & \mathsf{Dyn}\,\theta & <: & \theta \end{array}$$

Note that any closed term can be given a type of the form $\mathsf{Const}\,\theta$. For example, the counter class of Section 2 has the type Const (cls *counter*).

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

$$\frac{\Gamma \rhd C : \mathsf{cls}\,\tau \quad \Gamma, x{:}\tau \rhd M : \theta \quad \Gamma, x{:}\tau \rhd A : \mathsf{comm}}{\Gamma \rhd (\mathbf{class}\,\theta\ \mathbf{fields}\ C\ x\ \mathbf{methods}\ M\ \mathbf{init}\ A) : \mathsf{cls}\,\theta}$$
$$(\text{if } \Gamma \text{ is passively used in } A)$$

This allows the free identifiers Γ to be used in A, but in a read-only fashion. The parametricity interpretation of 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 (2) becomes conditional on non-interference:

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

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.

B Semantics of specifications

In this section, we consider the issue of interpreting specifications. This raises two issues. First, the non-interference formulas in specifications require a sophisticated functor category interpretation [57, 41] whose relationship to the parametricity interpretation is not yet well-understood. It is however possible to interpret restricted versions of specifications, those in which $\forall$-quantified identifiers are restricted not to interfere with any other free identifiers. Note that the queue specification in Fig. 3 is of this form. The second issue, discussed in Section 3.1, is that the equality relation of specifications must be general enough to be refined by implementations.

To allow for equality relations to be refined in implementations, we define a parametric PER semantics for IA$^+$. The basic ideas are from Bainbridge et al. [7]. (See also [8].) We adapt them to a predicative polymorphic context. A PER E over a set X is a symmetric and transitive relation. (It differs from an equivalence relation in that it need not be reflexive.) The *domain* of E is defined by $x \in \mathrm{dom}(E) \iff x\ E\ x$. Note that E reduces to a (total) equivalence relation over $\mathrm{dom}(E)$. The set of equivalence classes under E is denoted $Q(E)$. See [34, Sec. 5.6] for discussion of PER's.

A "type" in the new setting (called a PER-*type*) is a pair $X = \langle X, E_X \rangle$ where X is a set and E_X is a PER over X. The PER specifies the notion of "equality" for the type. A "relation" $R : \langle X, E_X \rangle \leftrightarrow \langle X', E_{X'} \rangle$ is an ordinary relatin $R : X \leftrightarrow X'$ that satisfies $E_X; R; E_{X'} = R$ (called a *saturated relation*).

A "type operator" is a pair $\langle T_{\mathrm{per}}, T_{\mathrm{rel}} \rangle$ of mappings for per-types and saturated relations. The PER-type operators for products and function spaces are as follows:

$$\begin{array}{l} \langle X, E_X \rangle \times \langle Y, E_Y \rangle = \langle X \times Y,\, E_X \times E_Y \rangle \\ \qquad\qquad R \times S = R \times S \\ \langle X, E_X \rangle \to \langle Y, E_Y \rangle = \langle X \to Y,\, E_X \to E_Y \rangle \\ \qquad\qquad [R \to S] = [E_X \to E_Y];\ [R \to S];\ [E_X \to E_Y] \end{array}$$

Assume that $\mathcal{S}$ is a small collection of PER-types. We are interested in PER-type operators over $\mathcal{S}$. These operators

$$
\begin{array}{rcl}
Q,\eta \models M =_\theta N & \iff & [\![M]\!]_Q\eta \; E_{[\![\theta]\!](Q)} \; [\![N]\!]_Q\eta \\
Q,\eta \models \{P\}A\{P'\} & \iff & \forall q,q' \in Q.\, [\![P]\!]_Q\eta q = \mathsf{true} \wedge [\![A]\!]_Q\eta q = q' \Longrightarrow [\![P']\!]_Q\eta q' = \mathsf{true} \\
Q,\eta \models \phi \Longrightarrow \phi' & \iff & \forall Z.\, (Q \times Z,\, \eta \uparrow_Q^{Q\times Z} \models \phi) \Longrightarrow (Q \times Z,\, \eta \uparrow_Q^{Q\times Z} \models \phi') \\
Q,\eta \models \forall x{:}\theta.\, (\&_i\, x \,\#\, x_i) \Longrightarrow \phi & \iff & \forall Z. \forall v \in \mathrm{dom}(E_{[\![\theta]\!](Z)}).(Q \times Z,\, \eta \uparrow_Q^{Q\times Z} [x \to v \uparrow_Z^{Q\times Z}]) \models \phi \\
Q,\eta \models \exists x{:}\theta.\, \phi & \iff & \exists v \in [\![\theta]\!](Q).\, (Q,\, \eta[x \to v] \models \phi) \\
Q,\eta \models \mathbf{Inst}\; C\; x.\, \phi & \iff & \exists \langle Z, \langle p, z_0\rangle\rangle \sim^* [\![C]\!]_Q\eta.(Q \times Z,\, \eta \uparrow_Q^{Q\times Z} [x \to p] \models \phi)
\end{array}
$$

Figure 8: Interpretation of specifications

inherit product, sum and function space constructors from the above notions. We define type quantifiers as follows:

- The PER-type operator $\forall Z.\, T(X,Z)$ maps a PER-type X to the PER-type $\langle \prod_{Z\in\mathcal{S}} T(X,Z),\, \forall^{\mathrm{sat}} S.\, T(E_X, S)\rangle$ The set consists of families indexed by $Z \in \mathcal{S}$. The PER equates two families p and p' if for all saturated relations $S\colon Z \leftrightarrow Z'$, we have $p_Z\; T(E_X,S)\; p'_{Z'}$. The relation part of the operator maps a saturated relation $R\colon X \leftrightarrow X'$ to $\forall^{\mathrm{sat}} S.\, T(R,S)$.
- The PER-type operator $\exists Z.\, T(X,Z)$ maps a PER-type X to the PER-type $\langle \sum_{Z\in\mathcal{S}} T(X,Z),\, \sim_X^+\rangle$ where $\sim_X$ is given by

  $$\langle Z,p\rangle \sim_X \langle Z',p'\rangle \iff \exists^{\mathrm{sat}} S\colon Z \leftrightarrow Z'.\, p\; T(E_X,S)\; p'$$

  The relation part of the operator maps a saturated relation $R\colon X \leftrightarrow X'$ to $\sim_X^+; \exists^{\mathrm{sat}} S.\, T(R,S); \sim_{X'}^+$.

Comparing this to the plain parametricity semantics of Section 3.1, we note that PER's take the place of the identity relations.

Theorem 5 *Every type operator $T(X_1,\ldots,X_n)$ maps* PER*-types to* PER*-types and saturated relations $R_i\colon X_i \leftrightarrow X_i'$ to saturated relations $T(R_1,\ldots,R_n)$. Further, $T(E_{X_1},\ldots,E_{X_n}) = E_{T(X_1,\ldots,X_n)}$.*

The proof is similar to that in [8].

The interpretation of IA^+ is exactly the same as in plain parametricity semantics except that the type operators are now understood to be PER-type operators. The interpretation of specifications is shown in Fig. 8. A judgment of the form $Q,\eta \models \phi$ means that the formula ϕ with free identifiers Γ *holds* in the state set Q and environment $\eta \in \mathrm{dom}([\![\Gamma]\!](E_Q))$.

References

[1] Abadi, M., and Cardelli, L. An imperative object calculus. *Theory and Practice of Object Systems 1*, 3 (1996), 151–166.

[2] Abadi, M., and Cardelli, L. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Abadi, M., and Leino, R. M. A logic of object-oriented programs. In *TAPSOFT '97 and CAAP/FASE*, vol. 1214 of *LNCS*. Springer-Verlag, 1997, pp. 682–696.

[4] Abramsky, S., and McCusker, G. Linearity, sharing and state. In *Algol-like Languages* [43], ch. 20.

[5] America, P. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds., vol. 489 of *LNCS*. Springer-Verlag, 1990, pp. 60–90.

[6] Andersen, D. S., Pedersen, L. H., Hüttel, H., and Kleist, J. Objects, types and modal logics. In *FOOL 4* (http://www.cs.indiana.edu/hyplan/pierce/fool/, 1997), Electronic proceedings.

[7] Bainbridge, E. S., Freyd, P., Scedrov, A., and Scott, P. J. Functorial polymorphism. *Theoretical Comput. Sci. 70* (1990), 35–64.

[8] Berlucci, R., Abadi, M., and Curien, P.-L. A model for formal parametric polymorphism: A PER interpretation for System R. In *Typed Lambda Calculi and Applications - TLCA '95*, LNCS. Springer-Verlag, 1995, pp. 32–46. (Expanded version in a Manuscript, 1995).

[9] Borba, P., and Goguen, J. Refinement of concurrent object-oriented programs. In *Formal Methods and Object Technology*, S. Goldsack and S. Kent, Eds. Springer-Verlag, 1996, ch. 11.

[10] Brookes, S., Main, M., Melton, A., and Mislove, M., Eds. *Mathematical Foundations of Programming Semantics: Eleventh Annual Conference*, vol. 1 of *Electronic Notes in Theor. Comput. Sci.* Elsevier, 1995.

[11] Bruce, K. B. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP'95*, vol. 952 of *LNCS*. Springer-Verlag, 1995, pp. 27–51.

[12] Bruce, K. C., Cardelli, L., and Pierce, B. C. Comparing object encodings. Tutorial at FOOL 3 (Manuscript available from authors), 1996.

[13] Cardelli, L. A semantics of multiple inheritance. In *Semantics of Data Types*, G. Kahn, D. B. MacQueen, and G. Plotkin, Eds., vol. 173 of *LNCS*. Springer-Verlag, 1984, pp. 51–67.

[14] Carlsson, M., and Hallgren, T. FUDGETS: A graphical user interface in a lazy functional language. In *FPCA '93: Conf. on Functional Program. Lang. and Comput. Arch.* (1993), ACM, pp. 321–330.

[15] Cook, W. R. *A Denotational Semantics of Inheritance*. PhD thesis, Dep. of Computer Science, Brown Univ., May 1989. (Tech. Report CS-89-33).

[16] Cook, W. R., Hill, W. L., and Canning, P. S. Inheritance is not subtyping. In *ACM Symp. on Princ. of Program. Lang.* (1990), ACM, pp. 125–135. (Reprinted as Chapter 14 of [23]).

[17] Dahl, O.-J., and Nygaard, K. An Algol-based simulation language. *Comm. ACM 9*, 9 (Sept. 1966), 671–678.

[18] DiBlasio, P., and Fisher, K. A calculus for concurrent objects. In *CONCUR '96*, vol. 1119 of *LNCS*. Springer-Verlag, 1996, pp. 655–670.

[19] EIFRIG, J., SMITH, S., TRIFONOV, V., AND ZWARICO, A. An interpretation of typed OOP in a language with state. *J. Lisp and Symbolic Comput. 8*, 4 (1995), 357–397.

[20] FISHER, K., AND MITCHELL, J. C. Classes = objects + data abstraction. Tech. Rep. STAN-CS-TN-96-31, Stanford University, 1996.

[21] FISHER, K., AND MITCHELL, J. C. On the relationship between classes, objects and data abstraction. In *Mathematics of Program Construction, Marktoberdorf summer school*, LNCS. Springer-Verlag, 1997, p. (to appear).

[22] GIRARD, J.-Y., LAFONT, Y., AND TAYLOR, P. *Proofs and Types.* Cambridge Univ. Press, 1989.

[23] GUNTER, C. A., AND MITCHELL, J. C., Eds. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, 1994.

[24] GUTTAG, J. V., HOROWITZ, E., AND MUSSER, D. R. Abstract data types and software validation. *Comm. ACM 21*, 12 (Dec. 1978), 1048–1063.

[25] HE, J. F., HOARE, C. A. R., AND SANDERS, J. W. Data refinement refined. vol. 213 of *LNCS*. Springer-Verlag, 1986, pp. 187–196.

[26] HOARE, C. A. R. Record handling. In *Programming Languages: NATO Advanced Study Institute*, F. Genuys, Ed. Academic Press, London, 1968, pp. 291–347.

[27] KINOSHITA, Y., O'HEARN, P. W., POWER, A. J., TAKEYAMA, M., AND TENNENT, R. D. An axiomatic approach to binary logical relations with applications to data refinement. Manuscript, Queen Mary and Westfield, London, 1997.

[28] LANO, K., AND HAUGHTON, H. Reasoning and refinement in object-oriented specification languages. In *ECOOP '97*, O. L. Madsen, Ed., vol. 615 of *LNCS*. Springer-Verlag, 1992, pp. 78–97.

[29] LEAVENS, G. T. Modular specification and verification of object-oriented programs. *IEEE Software* (July 1991), 72–80.

[30] LISKOV, B., AND WING, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (1994), 1811–1841.

[31] MASON, I. A., AND TALCOTT, C. L. Axiomatizing operational equivalence in the presence of side effects. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science* (June 1989), IEEE Computer Society Press, pp. 284–293.

[32] MEYER, A. R., AND SIEBER, K. Towards fully abstract semantics for local variables. In *Fifteenth Ann. ACM Symp. on Princ. of Program. Lang.* (1988), ACM, pp. 191–203. (Reprinted as Chapter 7 of [43]).

[33] MILNER, R. An algebraic definition of simulation between programs. In *Proc. Second Intern. Joint Conf. on Artificial Intelligence* (London, 1971), The British Computer Society, pp. 481–489.

[34] MITCHELL, J. C. *Foundations of Programming Languages.* MIT Press, 1997.

[35] MITCHELL, J. C., AND PLOTKIN, G. D. Abstract types have existential types. *ACM Trans. Program. Lang. Syst. 10*, 3 (1988), 470–502.

[36] MORGAN, C., AND VICKERS, T., Eds. *On the Refinement Calculus.* Springer-Verlag, 1992.

[37] ODERSKY, M., RABIN, D., AND HUDAK, P. Call by name, assignment and the lambda calculus. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* (1993), ACM.

[38] O'HEARN, P. W., POWER, A. J., TAKEYAMA, M., AND TENNENT, R. D. Syntactic control of interference revisited. In Brookes et al. [10]. (Reprinted as Chapter 18 of [43]).

[39] O'HEARN, P. W., AND REDDY, U. S. Objects, interference and Yoneda embedding. In Brookes et al. [10].

[40] O'HEARN, P. W., AND TENNENT, R. D. Semantics of local variables. In *Applications of Categories in Computer Science*, M. P. Fourman, P. T. Johnstone, and A. M. Pitts, Eds. Cambridge Univ. Press, 1992, pp. 217–238.

[41] O'HEARN, P. W., AND TENNENT, R. D. Semantical analysis of specification logic, Part 2. *Inf. Comput. 107*, 1 (1993), 25–57. (Reprinted as Chapter 14 of [43]).

[42] O'HEARN, P. W., AND TENNENT, R. D. Parametricity and local variables. *J. ACM 42*, 3 (1995), 658–709. (Reprinted as Chapter 16 of [43]).

[43] O'HEARN, P. W., AND TENNENT, R. D. *Algol-like Languages (Two volumes).* Birkhäuser, Boston, 1997.

[44] PIERCE, B. C., AND TURNER, D. N. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming 4*, 2 (Apr. 1994), 207–247.

[45] PLOTKIN, G., AND ABADI, M. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications - TLCA '93* (1993), LNCS, Springer-Verlag, pp. 361–375.

[46] REDDY, U. S. Objects as closures: Abstract semantics of object-oriented languages. In *ACM Symp. on LISP and Functional Program.* (July 1988), ACM, pp. 289–297.

[47] REDDY, U. S. Global state considered unnecessary: Semantics of interference-free imperative programming. In *ACM SIGPLAN Workshop on State in Program. Lang.* (June 1993), Technical Report YALEU/DCS/RR-968, pp. 120–135.

[48] REDDY, U. S. Passivity and independence. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science* (July 1994), IEEE Computer Society Press, pp. 342–352.

[49] REDDY, U. S. Global state considered unnecessary: An introduction to object-based semantics. *J. Lisp and Symbolic Computation 9* (1996), 7–76. (Reprinted as Chapter 19 of [43]).

[50] REYNOLDS, J. C. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.* (1978), ACM, pp. 39–46. (Reprinted as Chapter 10 of [43]).

[51] REYNOLDS, J. C. The essence of Algol. In *Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds. North-Holland, 1981, pp. 345–372. (Reprinted as Chapter 3 of [43]).

[52] REYNOLDS, J. C. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, D. Neel, Ed. Cambridge Univ. Press, 1982, pp. 121–161. (Reprinted as Chapter 6 of [43]).

[53] REYNOLDS, J. C. Types, abstraction and parametric polymorphism. In *Information Processing '83*, R. E. A. Mason, Ed. North-Holland, Amsterdam, 1983, pp. 513–523.

[54] SCHOETT, O. Behavioral correctness of data representations. *Sci. Comput. Programming 14* (1990), 43–57.

[55] STARK, I. Categorical models for local names. *Lisp and Symbolic Computation 9*, 1 (Feb. 1996), 77–107.

[56] SWARUP, V., REDDY, U. S., AND IRELAND, E. Assignments for applicative languages. In *Algol-like Languages* [43], ch. 9, pp. 235–272.

[57] TENNENT, R. D. Semantical analysis of specification logic. *Inf. Comput. 85*, 2 (1990), 135–162. (Reprinted as Chapter 13 of [43]).

[58] TENNENT, R. D. Denotational semantics. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds., vol. 3. Oxford University Press, 1994, pp. 169–322.

[59] WINSKEL, G. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 354 of *LNCS*. Springer-Verlag, 1989, pp. 364–397.