An Automata-Theoretic Model of Objects

Uday S. Reddy
University of Birmingham

Brian P. Dunphy
University of Illinois at Urbana-Champaign

***Abstract*—In this paper, we present a new model of class-based Algol-like programming languages inspired by automata-theoretic concepts. The model may be seen as a variant of the "object-based" model previously proposed by the author, where objects are described by their observable behaviour in terms of events. At the same time, it also reflects the intuitions behind state-based models studied by Reynolds, Oles, Tennent and O'Hearn where the effect of commands is described by state transformations. The idea is to view stores as automata, capturing not only their states but also the allowed state transformations. In this fashion, we are able to combine both the state-based and event-based views of objects. We illustrate the efficacy of the model by proving several test equivalences and discuss its connections to the previous models.**

I. Introduction

Imperative programming languages provide *information hiding* via local variables, which is exploited in object-oriented programming in a fundamental way. The use of such information hiding in everyday programming can be said to have revolutionized the practice of software development.

Meyer and Sieber [15] pointed out that the traditional semantic models for imperative programs do not capture such information hiding, even though researchers working in the area were probably aware of the issues much earlier. Rapid progress was made in the 1990's to address the problem. O'Hearn and Tennent [19] proposed a model using relational parametricity to capture the independence of data representations. Reddy [23] proposed an alternative event-based model which hides data representations entirely. Both the models have been proved fully abstract for second-order types of Idealized Algol (though this does not cover "passive" or "read-only" types) [16, 17]. Abramsky and McCusker [1], and subsequently with Honda [2], refined the event-based model using games semantics and proved it fully abstract for full higher-order types.

Despite all this progress, the practical application of these models for program reasoning had stalled. As we shall see, "second-order functions" in Idealized Algol only correspond to basic functions (almost zero-order functions) in the object-oriented setting. The event-based model is a bit removed from the normal practice in program reasoning, while the applicability of the parametricity model for genuine higher-order functions has not been investigated. In fact, Pitts and Stark [22] showed in "awkward example" in a bare bones ML-like language, which could not be handled using the parametricity technique.

The present work began in the late 90's with the motivation of bridging the gap between state-based parametricity models and the event-based models, because they clearly had complementary strengths. These investigations led to an automata theory-inspired framework where both states and events play a role [24, 27, 26]. However, it was noticed that the basic ingredients of the model were already present in the early work of Reynolds [30]. The subsequent work focused on formalizing the category-theoretic foundations of the framework, documented in Dunphy's PhD thesis [8, 9], but the applications of the framework remained unexplored.

The interest in the approach has been renewed with two parallel developments in recent work. Amal Ahmed, Derek Dreyer and colleagues [3, 6, 7] began to investigate reasoning principles for higher-order ML-like languages where similar ideas have reappeared. In the application of Separation Logic to concurrency, a technique called "deny-guarantee reasoning" has been developed [4, 5] where, again, a combination of states and events is employed. With this paper, we hope to provide a denotational semantic foundation for these techniques and stimulate further work in this area.

II. Motivation

In this section, we informally motivate the ideas behind the new semantic model.

The two existing classes of semantic models for imperative programs are state-based parametricity models, formulated by O'Hearn and Tennent [19] and event-based models, formulated by Reddy [23]. Both of them were first presented for Algol-like languages, and later adapted to object-oriented programs [28].

In the state-based model, an object is described as a state machine with

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

For example, a counter object with methods for reading the value and for incrementing the state can be semantically described by:

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

Here, *val* is given by a function of type $Q \to Int$ and the effect of *inc* is given by a function of type $Q \to Q$. (We are ignoring the issues of divergence and recursion.) An alternative state machine for counters using a different representation (negative numbers) is described by:

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

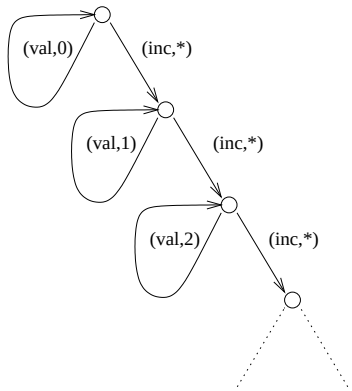

Fig. 1. Trace set of a counter object

The behavioral equivalence of the two implementations of counters can be established by exhibiting a *simulation relation* between the state sets:

$$n \left[R\right] n' \iff n \geq 0 \wedge n' = -n \tag{1}$$

and showing that all the operations "preserve" the simulation relation.

In contrast, the event-based description constructs a vocabulary of events for the methods of the object, e.g.,

$$\{\, (val, n) \mid n \in Int \,\} \cup \{(inc, *)\}$$

and describes the objects by their *trace sets*, i.e., sets of sequences of events that can be observed from the object. The trace set of counter objects can be depicted graphically, as shown in Fig. 1. The two distinct implementations of counter objects have exactly the *same* trace set in the event-based description. In this sense, the event-based description is more "extensional" than the state-based one. However, multiple traces that have the same effect on the internal state of the object are represented differently in the semantics.

A second, more subtle, difference between the two models is that the event-based description captures the *irreversibility* of state change. The action of incrementing the counter overwrites the old state of the counter and it is not possible to go back to the old state. For example, if we pass a counter object to a procedure, we can be sure that, after the procedure returns, the value in the counter could be no less that what it was before the call. This fact is obvious in the event-based description. The traces incorporate the direction of time. However, it is not possible to prove it in the state-based description. In short, the state-based description only captures what is the in the state but not how states are altered. Offsetting this technical deficiency, the state-based model has the advantage of being highly intuitive and quite familiar from traditional reasoning principles of programs.

In this paper, we define a new model that combines the advantages of the state-based and event-based models. For this purpose, we turn to automata theory. A *semiautomaton* in automata theory is a triple (Q, Σ, α), where Q is a set of states, Σ is a set of action symbols – representing *state transitions* – and a function $\alpha : \Sigma \times Q \to Q$ describing the effect of action symbols on states. The transition function is then extended to sequences of action symbols $\Sigma^* \times Q \to Q$. From this point of view, it is clear that the "state-based description" is focusing on state sets (Q) whereas the "event-based description" is focusing on the action symbols (Σ). A more abstract treatment of semiautomata is studied in algebraic automata theory [11, 13], called *transformation monoids*. The essential idea is to replace the free monoid of actions Σ^* by a monoid of state transformations $T \subseteq [Q \to Q]$. Such a monoid has an implicit action on states $\alpha : T \times Q \to Q$, viz., function application $\alpha(a, q) = a(q)$. An automata-theoretic model of objects can now be given in four parts:

- a state set Q,
- a monoid of state transformations $T \subseteq [Q \to Q]$,
- the initial state of the object, $q_0 \in Q$, and
- the effect of the methods on the object state as well as the state transformations.

For example, the automata-theoretic model of counter objects corresponding to M is:

$$\begin{aligned} N = \langle & Q = Int,\, T = Int^+, \\ & 0,\, \{val = \lambda n.\, n,\ inc = \lambda n.\, n + 1\}\rangle \end{aligned}$$

Here, $Int^+ = \{\, \lambda n.\, n + k \mid k \geq 0 \,\}$ is the set of allowed transformations that only increase the value of the internal state. Note that it is a monoid: $(\lambda n.\, n + k) \cdot (\lambda n.\, n + k') = \lambda n.\, n + (k + k')$. The type of val is $Q \to Int$ as before, but the type of inc is T. Any state change operations in methods are interpreted in T. So, they must be among the *allowed* transformations of the state machine.

The automata-theoretic model corresponding to M' is:

$$\begin{aligned} N' = \langle & Q' = Int,\, T' = Int^-, \\ & 0,\, \{val = \lambda n.\, -n,\ inc = \lambda n.\, n - 1\}\rangle \end{aligned}$$

Proving the equivalence of the two state machines requires us to exhibit *two relations*: a relation R_Q between the state sets and a relation R_T between the state transformations:

$$\begin{array}{lcl} n \left[R_Q\right] n' & \iff & n \geq 0 \wedge n' = -n \\ a \left[R_T\right] a' & \iff & \exists k.\, a = \lambda n.\, n + k \wedge a' = \lambda n.\, n - k \end{array}$$

The two relations have to satisfy some coherence conditions, which are detailed in Sec. IV. Using these relations, it is possible to prove, for instance, that a procedure that takes a counter as an argument can only increase the value of the counter (as visible from the outside). The transformation components in the state machines provide a direction of time, which is absent in the purely state-based model.

While simulation relations are useful for proving the equivalence of two implementations of classes, they form an instance of a general theory of relational parametricity which works for relations of arbitrary arity [12, 18]. The case of "unary relations" is particularly noteworthy because it gives us a new notion of *invariants*. Our theory therefore posits that invariants of classes again come in two parts: one on state sets and one

on state transformations. The invariants for counter objects represented by N are:

$$\begin{array}{lcl} P_Q(n) & \iff & n \geq 0 \\ P_T(a) & \iff & \exists k.\, a = \lambda n.\, n + k \end{array}$$

State invariants are well-known from traditional reasoning methods, while the invariant properties of transformations might be called "action invariants" or "transition invariants".

The recent work on reasoning about state has focused on higher-order procedures and higher-order state, in particular the work of Ahmed, Dreyer and colleagues [3, 6]. This work has brought home the fact that the traditional theory of Algol-like languages fails to be abstract for higher-order procedures.[1] We illustrate the problem with an example from Pitts and Stark [22], which was termed an "awkward example" in their paper. Consider the following class, written in the IA+ language [28]:

$$\begin{array}{l} C = \mathbf{class} : \mathbf{comm} \to \mathbf{comm} \\ \qquad \mathbf{local}\ \mathbf{var}[\mathrm{int}]\ x; \\ \qquad \mathbf{init}\ x := 0; \\ \qquad \mathbf{meth}\ \{m = \lambda c.\, x := 1; c; \mathit{test}(x = 1)\} \\ \mathit{test}(b) \triangleq \mathbf{if}\ b\ \mathbf{then\ skip\ else\ diverge} \end{array}$$

This class provides a single method of type $\mathbf{comm} \to \mathbf{comm}$, i.e., a procedure that takes a command-typed argument. (This is a call-by-name language, where commands can be passed as arguments, but similar examples can be constructed using call-by-value as well.) The problem is to argue that the method always terminates. Intuitively, one might expect that this should always be the case because the local variable x is only available inside the class. However, the intuition is not a very good guide here because the methods are higher-order. When the method of C invokes the argument command c, it is possible for c to alter x. For example, the following client does so:

$$\mathbf{new}\ C\ \lambda p.\, p.m\ (p.m\ \mathbf{skip})$$

When the outer call to $p.m$ is executed, it sets x to 1 and calls its argument $c \equiv p.m\ \mathbf{skip}$. Since the argument in turn calls $p.m$, it has the effect of setting x to 1. So, the argument that c does not have "access" to x is not sound.[2]

A more sophisticated argument for the termination of C's method notes that the only change that a call to c can make to x is setting it to 1. Therefore, at the end of the call to c, x is still 1, and so the test should succeed. However, as noted by Dreyer et al. [6], this cannot be proved by the usual "invariant-based" reasoning, i.e., by exhibiting relations on states. Instead, one must use relations on state transformations.

[1]Both the state-based and the event-based models have been proved fully abstract for second-order Algol types. Translated to object-oriented languages, this amounts to saying that they can prove the equivalence of classes whose methods take at best value-typed, i.e., state-independent, arguments. If the methods take higher-type arguments, e.g., other procedures, then the full abstraction results do not apply.

[2]Note that the system of "Syntactic Control of Interference" studied in the original object-based model [23] prohibits calls such as $p.m\ (p.m\ \mathbf{skip})$ because the procedure and the argument interfere. So, the naive intuition is sound for Syntactic Control of Interference. But it is not sound for full Idealized Algol.

In our framework, we start by defining a two-part invariant for the class:

$$\begin{array}{lcl} P_Q(x) & \iff & x = 0 \vee x = 1 \\ P_T(a) & \iff & a = (\lambda n.\, n) \vee a = (\lambda n.\, 1) \end{array}$$

To maintain P_T as an "invariant", the method m must restrict its actions to those satisfying P_T, while *assuming* that the argument c does so as well. So, by assumption, the call to c will either leave x unchanged or set it to 1. In either case, the value of x at the end of c will be 1. So, the method always terminates.

Other examples discussed by Dreyer et al. [6] can be verified similarly, as long as they fit within our framework — with only ground-typed state and no control effects.

III. Preliminaries

The programming language we use in this paper is the language IA+ described in [28], which represents Idealized Algol [30] extended with classes.

Recall that Idealized Algol is a call-by-name simply typed lambda calculus with base types supporting imperative programming. These base types include

$$\mathbf{val}[\delta] \qquad \mathbf{exp}[\delta] \qquad \mathbf{comm}$$

where δ ranges over "data types" such as $\mathbf{int}$ and $\mathbf{bool}$.

To support classes, we use a type constructor $\mathbf{cls}$ so that $\mathbf{cls}\theta$ is the type of classes whose method suite is of type θ. So, θ is the interface type of the class. The language comes with a family of predefined classes $\mathbf{var}[\delta]$ for assignable variables of type δ, whose type is

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

In essence, a variable is treated as an object with a "get" method that reads the state of the variable and "put" method that changes the state to a given value. User-defined classes are supported using terms of the form

$$\begin{array}{l} \mathbf{class} : \theta \\ \quad \mathbf{local}\ C\ x; \\ \quad \mathbf{init}\ A; \\ \quad \mathbf{meth}\ M \end{array}$$

where C is another class, x is a locally bound identifier for the "instance variable," A is a command for initializing the instance variable, and M is a term of type θ serving as the method suite. For simplicity of exposition, we only consider "constant classes" in the main body of the paper, which are defined by closed terms of type $\mathbf{cls}\ \theta$. See Appendix for a treatment of general classes with free identifiers.

Instances of classes are created in commands using terms of the form

$$\mathbf{new}\ C\ \lambda o.\, B$$

whose effect is to create an instance of class C, bind it to o and execute a command B where o is allowed to occur as a free identifier. So, $\mathbf{new}$ is a constant of type:

$$\mathbf{new} : \mathbf{cls}\ \theta \to (\theta \to \mathbf{comm}) \to \mathbf{comm}$$

Semantics of Algol-like languages is normally given using a category-theoretic possible world semantics, where the "worlds" represent the shapes of store. Each type θ is interpreted as a functor

$$[\![\theta]\!] : \mathbf{W}^{\mathrm{op}} \to \mathbf{CPO}$$

where $\mathbf{W}$ is a category of possible worlds and $\mathbf{CPO}$ is the category of directed-complete partial orders and continuous functions. So $[\![\theta]\!](W)$, for each world W, is a cpo. Further, if $f : X \to W$ is a morphism in the category of worlds, representing the idea that X is a "future wold" of W, then $[\![\theta]\!](f) : [\![\theta]\!](W) \to [\![\theta]\!](X)$ is a continuous function that restates values of type θ at world W as values of type θ at the future world X.[3] Note that the morphisms in the category of worlds go from future worlds to the current world. This is consistent with an object-oriented view because the future world generally represents a *larger* state than the current world. It is possible to think of a subtyping relationship $X <: W$ between the future world and the current world, except that there may not be a unique way in which X is a subtype of W. The morphism describes the way in which X is a subtype of W.

To incorporate relational parametricity, we extend categories with relations so that we formally work in *reflexive graphs* of categories [19, Sec. 7]. Intuitively, this means that we use two-dimensional categorical structures, where morphisms occupy one dimension and relations between categorical objects occupy the second dimension, as in the diagram below:

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[d, leftrightarrow, "R"'] & Y \arrow[d, leftrightarrow, "S"] \\ X' \arrow[r, "f'"] & Y' \end{tikzcd}$$

A diagram of this form, called a relation-preservation square, states the property that the morphisms f and f' map R-related arguments to S-related results. The textual notation for the property is $f \ [R \to S] \ f'$.

The reflexive graphs we work with are called *parametricity graphs* [8, 9]. They incorporate additional axioms to capture the idea that relations in the vertical dimension indeed behave like "relations" in the intuitive sense. The term *op-parametricity graph* is used to describe the structure $\mathbf{W}$ whose dual, $\mathbf{W}^{\mathrm{op}}$, is a parametricity graph. Our possible worlds form an op-parametricity graph.

The term *PG-functor* is used to denote maps between parametricity graphs. It involves a pair of functors, one for the category of objects and morphisms, and the other for the category of relations and relation-preservation squares. So, the overall structure of our interpretation is

$$[\![\theta]\!] : \mathbf{W}^{\mathrm{op}} \to \mathbf{CPO}$$

[3]In [19, Sec. 3] and [28], a "category-free" presentation of the semantic model is given to make it easier to follow. We do not yet know if a similar presentation is possible for our model. So, we use a full categorical treatment.

where $\mathbf{W}$ is an op-parametricity graph, $\mathbf{CPO}$ is a parametricity graph of cpo's (with directed-complete relations as relations) and $[\![\theta]\!]$ is a PG-functor. For the category of such PG-functors to be cartesian-closed, there is an additional requirement that the PG-functors should factor through the embedding of $\mathbf{CPO}_\perp$ in $\mathbf{CPO}$:

$$\mathbf{W}^{\mathrm{op}} \longrightarrow \mathbf{CPO}_\perp \longrightarrow \mathbf{CPO}$$

which means basically that each $[\![\theta]\!](W)$ should be a *pointed* cpo, each $[\![\theta]\!](f)$ should be a *strict* continuous function, and each $[\![\theta]\!](R)$ should be (*pointed*) complete relation.

The focus of this paper is on defining a suitable category (or, rather, an op-parametricity graph) $\mathbf{W}$. This is where automata-theoretic ideas come in. Defining the semantics itself follows more or less along the traditional lines [19, 16].

IV. Transformation monoids

Our starting point is the notion of transformation monoids that comes from algebraic automata theory.

A *semiautomaton* (or generically a "state machine") is usually represented as a triple (Q, Σ, α) where Q is a set (of "states"), Σ is a set (of "actions"), $\alpha : \Sigma \to [Q \rightharpoonup Q]$ is a function interpreting each action as a state-transformation function. A semiautomaton differs from a normal automaton in that it does not specify a start state or final states. It describes the generic behaviour of a state machine rather than a particular instance of the machine.

A semiautomaton can be viewed as a *representation* of the free monoid Σ^* by transformations of Q (or equivalently, as an *action* of the free monoid Σ^* on Q). A more abstract representation can be achieved by moving from free monoids Σ^* to arbitrary monoids T, which are still meant to represent "actions." Since all that matters about the elements of T is the action they have on the state set, we can as well take T to be a submonoid of the monoid of transformations $T(Q) = [Q \rightharpoonup Q]$. This leads to the concept of a *transformation monoid*.

a) Notation: We regard partial functions from A to B as *total* functions from A to $B_\perp$, but continue to use the notation $f : A \rightharpoonup B$ for such functions. The set of such functions $[A \rightharpoonup B]$ forms a directed complete partial order (dcpo) under the pointwise order (has sups of directed sets) and is pointed (has a least element). The $\rightharpoonup$ notation is also extended to functions and relations:

$$\begin{array}{lrcl} \text{(sets)} & [A \rightharpoonup B] & = & [A \to B_\perp] \\ \text{(functions)} & [f \rightharpoonup g] & = & [f \to g_\perp] \\ \text{(relations)} & [R \rightharpoonup S] & = & [R \to S_\perp] \end{array}$$

Recall that $g_\perp$ is the extension of a function $g : B \to B'$ to a strict function $B_\perp \to B'_\perp$ sending $\perp$ to $\perp$. Likewise, $S_\perp = S \cup \{(\perp, \perp)\}$. We write the sequential composition of "partial functions" as $f \cdot g$ or as $f; g^*$. When $x \in A_\perp$ and $y \in B_\perp$, we use the notation:

$$[x, y]_\perp \triangleq \begin{cases} \perp, & \text{if } x = \perp \text{ or } y = \perp \\ (x, y), & \text{otherwise} \end{cases}$$

Recall that a *monoid* is a set with an associative binary operation, denoted "$\cdot$", and a unit element for this operation.

The set of all (finite) sequences over a set Σ forms a monoid Σ^* with concatenation as the binary operation and the empty sequence serving as the unit element. This is in fact the free monoid generated by the set Σ. The set of all transformations $T(Q) = [Q \rightharpoonup Q]$ forms a monoid under sequential composition "$\cdot$" with the unit being the injection $j_Q : Q \rightharpoonup Q$. We also write the unit element as null_Q for better readability. In addition to being a monoid, $T(Q)$ is a pointed cpo (in fact a bounded complete cpo) under the pointwise ordering with the least element being the constantly-$\bot$ function $\overline{\bot}$.

We use the term *complete ordered monoid* to refer to a monoid that is also a pointed cpo, and the multiplication is a strict, continuous function. A *complete ordered submonoid* (or simply a "submonoid" when the context is clear) is a subset that is not only closed under the unit and multiplication but also contains the least element and the sup's of directed sets. A *morphism of complete ordered monoids* is a monoid morphism that is also strict and continuous.

A *transformation monoid* (tm) is a pair (Q, T) where Q is a set (of "states") and T is a submonoid of $T(Q)$. There is an implicit monoid action $\alpha : T \to [Q \rightharpoonup Q]$ coming from the fact that T is a submonoid of the monoid of transformations, which is a morphism of complete ordered monoids. We denote the uncurried function corresponding to α by $\alpha^\dagger : T \times Q \rightharpoonup Q$, which represents a monoid action.

A *relation of tm's* $R : (Q, T) \leftrightarrow (Q', T')$ is a pair $R = (R_Q, R_T)$ where $R_Q : Q \leftrightarrow Q'$ is a relation and $R_T : T \leftrightarrow T'$ is a complete ordered monoid relation (relation compatible with the units, multiplication, least elements and sup's of directed sets) such that they are compatible with the monoid action, i.e., $\alpha \left[R_T \to [R_Q \rightharpoonup R_Q]\right] \alpha'$. When there is no cause for confusion, we omit the subscripts in R_Q and R_T, e.g., we may write $q \left[R\right] q'$ for states and $a \left[R\right] a'$ for actions, using the context to distinguish the uses. The identity relation $I_{(Q,T)} : (Q, T) \leftrightarrow (Q, T)$ is $I_{(Q,T)} = (\Delta_Q, \Delta_T)$ consisting of the diagonal relations (equality relations) on states and transformations.[4]

Intuitively, the transformations in T represent the actions executed by commands in the programming language. The sequential composition "$\cdot$" corresponds to the sequential composition in the programming language and the unit transformation null_X represents the do-nothing command **skip**. So, it is reasonable to require that T should be closed under these operations and the relations between worlds should be compatible with them as well. However, while these operations were perhaps adequate for traditional automata theory, they are not enough to capture the computations in programming languages. A command in a programming language can read the information from the initial state and tailor its actions based on that information. Such state-dependent actions need to be represented by an operation in the transformation monoids.

Reynolds [30] noticed the problem and proposed an operation called the "diagonal" operation. We write it as "read_X" with the type $(Q \to T) \to T$ because it has the effect of "reading" the initial state. It has a straightforward definition:

$$\mathrm{read}_X(p) = \lambda x.\, p(x)(x)$$

The intuition is that given a state-dependent action p, $\mathrm{read}_X(p)$ executes it using the initial state to satisfy the state dependence of p.

We call a transformation monoid $X = (\mathcal{Q}_X, \mathcal{T}_X)$ that is closed under the Reynolds diagonal operation a *Reynolds transformation monoid* (or "rtm" for short). A relation of rtm's is a relation of tm's $R = (R_Q, R_T)$ that is compatible with the Reynolds diagonal operation:

$$\mathrm{read}_X \left[[R_Q \to R_T] \to R_T\right] \mathrm{read}_{X'}$$

If $X = (\mathcal{Q}_X, \mathcal{T}_X)$ is a transformation monoid that is not closed under the read_X operation, then additional elements can be added to $\mathcal{T}_X$ so that it becomes closed under read_X. The *read closure* of $\mathcal{T}_X$ is the least set of transformations $\mathcal{R}(\mathcal{T}_X)$ closed under read_X. Such a closure is guaranteed to exist because $T(\mathcal{Q}_X)$ is always read-closed, and read-closure is preserved under intersection by the usual argument of universal algebra.

Examples of relations

Transformation monoids place an upper bound on the transformation components of relations R_T (which should be included in $[R_Q \rightharpoonup R_Q]$). But there is no lower bound other than the trivial one: $\{(j_X, j_{X'}), (\overline{\bot}, \overline{\bot})\}$. Reynolds transformation monoids require R_T to be closed under the read operations, placing a requirement on what should be included in R_T.

1) It is always permissible to pick R_T to be $[R_Q \rightharpoonup R_Q]$ for any given state relation R_Q. If $p \left[R_Q \to [R_Q \rightharpoonup R_Q]\right] p'$ then clearly $s \left[R_Q\right] s'$ implies $p(s)(s) \left[(R_Q)_\bot\right] p'(s')(s')$. Hence, $\mathrm{read}_X(p)(s) \left[R_Q \rightharpoonup R_Q\right] \mathrm{read}_X(p)(s)$. However, this choice of R_T means that we are not using the additional degree of freedom available in the transformation components of tm's.
2) For a more interesting example, consider the relation $R : (Int, T(Int)) \leftrightarrow (Int, T(Int))$ defined by:

$$\begin{aligned} n \left[R\right] n' &\iff n = n' \\ a \left[R\right] a' &\iff a = a' \wedge (\forall n.\, a(n) = \bot \vee a(n) \geq n) \end{aligned}$$

The relation is really a "unary" relation (or "invariant") represented in binary form. While the state part of the invariant is unconstrained, the transformation part states that the integer value of the state can only increase during command execution. Such a constraint may be thought of as a "transition invariant" or "step invariant." To check that it preserves the read operation, suppose p is related to itself by $R_Q \to R_T$, i.e., for all states n, $p(n)$ is related to itself by R_T. Then $p(n)(n)$ is either $\bot$ or a larger integer than n. So, $\mathrm{read}(p)$ is related to itself by R_T.

[4]Since the transformation components of tm's are posets, it is also possible to use the partial order $\sqsubseteq_T$ of the transformations as the second component of identity edges. This choice would lead to a possible world category with a similar effect to that of Tennent [33].

3) As a binary version of the above example, consider $R : (Int, T(Int)) \leftrightarrow (Int, T(Int))$ defined by:

$$\begin{array}{l} n \left[R\right] n' \iff n \geq 0 \wedge n' = -n \\ a \left[R\right] a' \iff \forall n, n'.\, a(n) - n \simeq -(a'(n') - n') \end{array}$$

This relates transformations a and a' whenever a increases the integer state by some amount and a' decreases the state by the same amount.[5]

4) As a trivial example, consider $R = (R_Q, R_T)$ where R_Q is arbitrary and $R_T = \{\, (a, a') \mid a \sqsubseteq j_X \wedge a' \sqsubseteq j_{X'} \,\}$. Then, assuming $p \left[R_Q \to R_T\right] p'$, and $s \left[R_Q\right] s'$, we have $p(s) \sqsubseteq j_X$ and $p'(s') \sqsubseteq j_{X'}$, which implies $\mathrm{read}_X(p) \sqsubseteq j_X$ and $\mathrm{read}_{X'}(p') \sqsubseteq j_{X'}$.

The following result is technical, but it gives some intuition for the strength of the read-closed condition.

Lemma 1 (Down-closure) 1) In an rtm $(\mathcal{Q}_X, \mathcal{T}_X)$ the transformation component is down-closed, i.e., $a \in \mathcal{T}_X$ and $a' \sqsubseteq a$ implies $a' \in \mathcal{T}_X$.

2) If $R : (\mathcal{Q}_X, \mathcal{T}_X) \leftrightarrow (\mathcal{Q}_Y, \mathcal{T}_Y)$ is a relation of rtm's, the transformation component R_T is "parallel down-closed," i.e., $a \left[R_T\right] b$, $(a', b') \sqsubseteq (a, b)$ and $a' \left[R_Q \rightharpoonup \mathbb{T}\right] b'$ implies $a' \left[R_T\right] b'$. ($\mathbb{T}$ is the universally true relation.)

Morphisms

We will designate some of the relations of rtm's as "morphisms" so that they can be used to talk about possible worlds.

Note that, whenever $f : A \to A'$ is a set-theoretic function, its function graph is a binary relation $\langle f \rangle : A \leftrightarrow A'$. If $R : A \leftrightarrow A'$ is a relation, we write $R^{\smile} : A' \leftrightarrow A$ for the converse of R.

A *morphism of rtm's* $f : X \to W$ is a pair $f = (\phi_f, \tau_f)$ where $\phi_f : \mathcal{Q}_X \to \mathcal{Q}_W$ is a function and $\tau_f : \mathcal{T}_W \to \mathcal{T}_X$ is a complete ordered monoid morphism such that the pair $(\langle \phi_f \rangle, \langle \tau_f \rangle^{\smile})$ is a relation of rtm's.

$$\begin{array}{c} X \\ \downarrow f \\ W \end{array} \quad = \quad \left(\begin{array}{c} \mathcal{Q}_X \\ \downarrow \phi_f \\ \mathcal{Q}_W \end{array} \;,\; \begin{array}{c} \mathcal{T}_X \\ \uparrow \tau_f \\ \mathcal{T}_W \end{array} \right)$$

Computationally, the intuition is that, when X is a future world of W, it *extends* and possibly *constrains* the states of the current world. So, it is possible to recover the state information at the level of the current world W via the function ϕ_f. On the other hand, the actions possible in the current world continue to be possible in the future world, which is modelled by the function τ_f going in the *opposite* direction.

The condition that $(\langle \phi_f \rangle, \langle \tau_f \rangle^{\smile})$ is a relation of rtm's amounts to the following properties:

- τ_f is a morphism of complete ordered monoids, i.e., it is a strict, continuous function that preserves the unit and the composition.
- the implicit monoid action is preserved:

$$\alpha_X \left[\langle \tau_f \rangle^{\smile} \to [\langle \phi_f \rangle \rightharpoonup \langle \phi_f \rangle]\right] \alpha_W$$

which can be expressed more directly by writing

$$\forall a \in \mathcal{T}_W .\, (\phi_f)_\perp \circ \tau_f(a) = a \circ \phi_f$$

- the Reynolds diagonal operation is preserved:

$$\mathrm{read}_X \left[[\langle \phi_f \rangle \to \langle \tau_f \rangle^{\smile}] \to \langle \tau_f \rangle^{\smile}\right] \mathrm{read}_W$$

which can be written equivalently as, for all $p \in (\mathcal{Q}_W \to \mathcal{T}_W)$,

$$\tau_f(\mathrm{read}_W(p)) = \mathrm{read}_X(\tau_f \circ p \circ \phi_f)$$

The knowledgeable reader will be able to verify that these are precisely the morphisms considered by Reynolds [30], except that he used full transformation monoids where $\mathcal{T}_X$ is always $T(\mathcal{Q}_X)$.

A *relation-preservation square* of rtm's

$$\begin{array}{ccc} (\mathcal{Q}_X, \mathcal{T}_X) & \xrightarrow{f = (\phi_f, \tau_f)} & (\mathcal{Q}_W, \mathcal{T}_W) \\ (S_Q, S_T) \updownarrow & & \updownarrow (R_Q, R_T) \\ (\mathcal{Q}_{X'}, \mathcal{T}_{X'}) & \xrightarrow{f' = (\phi_{f'}, \tau_{f'})} & (\mathcal{Q}_{W'}, \mathcal{T}_{W'}) \end{array}$$

exists iff $\phi_f \left[S_Q \to R_Q\right] \phi_{f'}$ and $\tau_f \left[R_T \to S_T\right] \tau_{f'}$.

This data constitutes a cpo-enriched reflexive graph $\mathbf{RTM}$. (See [9, 19] for the background on reflexive graphs.) The partial order on morphisms $f, f' : X \to Y$ is given by:

$$f \sqsubseteq_{X \to Y} f' \iff \phi_f = \phi_{f'} \wedge \tau_f \sqsubseteq \tau_{f'}$$

To the best of our knowledge, these kinds of morphisms and relations between transformation monoids have not been studied in algebraic automata theory. The morphisms considered there generally keep the monoid of actions fixed, whereas our interest is in varying the monoid as well as the state set.

Lemma 2 $\mathbf{RTM}$ is a cpo-enriched op-parametricity graph, i.e., it is relational, op-fibred and satisfies the identity condition.

$\mathbf{RTM}$ is evidently relational. For op-fibration, we need a strongest post-edge $R[f, f']$ for every R, f and f' as in the situation shown below:

$$\begin{array}{ccc} Y & \xrightarrow{f} & X \\ R \updownarrow & & \uparrow\!\downarrow R[f, f'] \\ Y' & \xrightarrow[f']{} & X' \end{array}$$

We define it as the pair

$$R[f, f'] = (R_Q[\phi_f, \phi_{f'}],\ [\tau_f, \tau_{f'}]R_T)$$

[5]We use $\simeq$ to denote the Kleene equality: either both the sides are $\perp$ or both are defined and they are equal.

Diagrammatically:

$$\begin{tikzcd} \mathcal{Q}_Y \arrow[r, "\phi_f"] \arrow[d, leftrightarrow, "R_Q"'] & \mathcal{Q}_X \arrow[d, leftrightarrow, dashed] \\ \mathcal{Q}_{Y'} \arrow[r, "\phi_{f'}"'] & \mathcal{Q}_{X'} \end{tikzcd} \qquad \begin{tikzcd} \mathcal{T}_Y \arrow[d, leftrightarrow, "R_T"'] & \mathcal{T}_X \arrow[l, "\tau_f"'] \arrow[d, leftrightarrow, dashed] \\ \mathcal{T}_{Y'} & \mathcal{T}_{X'} \arrow[l, "\tau_{f'}"] \end{tikzcd}$$

The first component is the strongest post-edge in $\mathbf{Set}$ which is nothing but the "direct image":

$$R_Q[\phi_f, \phi_{f'}] = \{\, (x, x') \mid \exists y, y'.\, y \left[R_Q\right] y' \wedge \phi_f(y) = x \wedge \phi_{f'}(y') = x' \,\}$$

and the second component is the weakest pre-edge in the reflexive graph of complete ordered monoids which is nothing but the "inverse image":

$$[\tau_f, \tau_{f'}]R_T = \{\, (b, b') \mid \tau_f(b) \left[R_T\right] \tau_{f'}(b') \,\}$$

■

An op-parametricity graph has a *subsumption* map whereby each morphism $f : Y \to X$ is "subsumed" by a relation $\langle f \rangle : Y \leftrightarrow X$. This is given by $\langle f \rangle = I_Y[\mathrm{id}_Y, f]$. In the case of $\mathbf{RTM}$, this gives $\langle(\phi_f, \tau_f)\rangle = (\langle \phi_f \rangle, \langle \tau_f \rangle^{\smile})$.

Examples of morphisms

1) The *expansion* of a full transformation monoid $(Q, T(Q))$ with additional state components represented by a set Z, and leading to a larger world $(Q \times Z, T(Q \times Z))$, is represented by a morphism $\times Z = (\phi, \tau) : (Q \times Z, T(Q \times Z)) \to (Q, T(Q))$. Here, $\phi : Q \times Z \to Q$ is the projection of the Q component, and $\tau : T(Q) \to T(Q \times Z)$ is given by

$$\tau(a)(q, z) = [a(q), z]_\perp = (a(q) = \perp \to \perp;\ (a(q), z))$$

This example is from [30], and it is easy to verify that $\times Z$ preserves the implicit monoid action and the Reynolds diagonal.

2) A *state change restriction* morphism for a tm $(\mathcal{Q}_X, \mathcal{T}_X)$ restricts the state transformations to a submonoid $T' \subseteq \mathcal{T}_X$. The morphism $f = (\phi, \tau) : (\mathcal{Q}_X, \mathcal{T}_X) \to (\mathcal{Q}_X, T')$ is given by $\phi = \mathrm{id}_{\mathcal{Q}_X}$ and τ the injection of T' in $\mathcal{T}_X$.
3) A *passivity restriction* morphism is an extreme case of state change restriction morphism that prohibits all state changes: $p_X = (\mathrm{id}_{\mathcal{Q}_X}, \tau) : (\mathcal{Q}_X, \mathcal{T}_X) \to (\mathcal{Q}_X, \mathbf{0}_X)$ where $\mathbf{0}_X$ is the complete ordered monoid containing the unit transformation $j_{\mathcal{Q}_X}$ and all its approximations.

Note that "state set restriction" and "state change constraints" morphisms found in the Tennent's category of worlds [33] do not have any counterparts in $\mathbf{RTM}$.

V. Modeling Stores

Now that we have the basic definitions of transformation monoids, we would like to present the intuition that they model stores of locations viewed as a rudimentary form of objects. This view point fits somewhere in between the state-based models [17, 19], where stores are viewed in a static form as sets of states, and the object-based models [16, 28], where stores are viewed as full-blown objects. Compared to the state-based models, we have more "activity" represented in transformation monoids. The allowed state transformations are part of the descriptions. Compared to the object-based models, we have less "activity." Only the state transformation aspects of the objects are retained in the description.

Nevertheless, the intuitions to be used for understanding the transformation monoids are similar to those of the object-based model. A morphism $f : X \to W$ may be thought of as a way of constructing a W-typed object from an X-typed object. In doing so, all the states of W should be representable in the X-typed store. Moreover, all the state transformations needed for W should be allowed on the X-typed store. For example, let X be a store representing a single integer variable and let W be a store representing a counter object. Then X allows all possible transformations of the integer state, whereas W needs only the transformations corresponding to incrementing the counter. Since the latter is a subset of the former, we have a morphism $X \to W$ (a state change restriction morphism), but there is no morphism in the opposite direction.

These intuitions come into the fore in trying to define "products" of transformation monoids. Suppose $X = (\mathcal{Q}_X, \mathcal{T}_X)$ and $Y = (\mathcal{Q}_Y, \mathcal{T}_Y)$ are rtm's denoting two separate stores of locations (along with allowed transformations). We would like to define a product rtm $X \star Y$ that corresponds to their combined store. There are two separate ways of doing this, depending on what transformations are allowed on the combined store. The "independent product", denoted $X \otimes Y$, allows the two parts of the store to be used independently, with no transfer of information between them. The "dependent product", denoted $X \star Y$, allows information to be transferred between them.

Independent product

Given transformations $a \in \mathcal{T}_X$ and $b \in \mathcal{T}_Y$, we use the notation $a \otimes b$ for the transformation in $T(\mathcal{Q}_X \times \mathcal{Q}_Y)$ defined by:

$$(a \otimes b)(x, y) = [a(x), b(y)]_\perp$$

Let $\mathcal{T}_X \otimes \mathcal{T}_Y$ denote the monoid of all transformations of the form $a \otimes b$. Then, the independent product of X and Y is defined as

$$X \otimes Y = (\mathcal{Q}_X \times \mathcal{Q}_Y,\ \mathcal{T}_X \otimes \mathcal{T}_Y)$$

The corresponding action on relations maps $R : X \leftrightarrow X'$ and $S : Y \leftrightarrow Y'$ to $R \otimes S : X \otimes Y \leftrightarrow X' \otimes Y'$ given by:

$$R_T \otimes S_T = \{\, (a \otimes b,\ a' \otimes b') \mid a \left[R_T\right] a' \wedge b \left[S_T\right] b' \,\}$$
$$R \otimes S = (R_Q \times S_Q,\ R_T \otimes S_T)$$

The $R_T \otimes S_T$ relation is well-defined even though $a \otimes b$ does not uniquely determine a and b.

The independent product has projections $(\pi_1)_{X,Y} : X \otimes Y \to X$ and $(\pi_2)_{X,Y} : X \otimes Y \to X$. For example,

$$(\pi_1)_{X,Y} = (\pi_1 : \mathcal{Q}_X \times \mathcal{Q}_Y \to \mathcal{Q}_X,\ \iota_1 : \mathcal{T}_X \to \mathcal{T}_X \otimes \mathcal{T}_Y) \text{ where } \iota_1(a) = a \otimes j_{\mathcal{Q}_Y}$$

It is easy to see that $(\pi_1)_{X,Y} \left[R \otimes S \to R\right] (\pi_1)_{X',Y'}$.

Dependent product

The dependent product $X \star Y$ is defined by:

$$\begin{array}{rcl} \mathcal{T}_X \star \mathcal{T}_Y & = & \text{the read-closure of } \mathcal{T}_X \otimes \mathcal{T}_Y \\ X \star Y & = & (\mathcal{Q}_X \times \mathcal{Q}_Y,\, \mathcal{T}_X \star \mathcal{T}_Y) \end{array}$$

While $\mathcal{T}_X \otimes \mathcal{T}_Y$ represents an independent product of the two stores, its read-closure adds transformations of the form $\lambda(x,y).\, a(x,y) \otimes b(x,y)$, allowing transfer of information between the two stores.

The corresponding relational action $R \star S : X \star Y \leftrightarrow X' \star Y'$ is a bit involved. The state set component is the expected one: $(R \star S)_Q = R_Q \times S_Q$. The transformation component is defined as follows:

$$\begin{array}{l} t \left[(R \star S)_T\right] t' \iff \\ \quad \forall x, x', y, y'.\ (x,y) \left[R \star S\right] (x', y') \Longrightarrow \\ \quad\quad (\exists a \in \mathcal{Q}_X,\, a' \in \mathcal{Q}_{X'},\, b \in \mathcal{Q}_Y,\, b' \in \mathcal{Q}_{Y'}. \\ \quad\quad\quad a \left[R\right] a' \wedge b \left[S\right] b' \wedge \\ \quad\quad\quad t(x,y) = (a \otimes b)(x,y) \wedge \\ \quad\quad\quad t'(x',y') = (a' \otimes b')(x',y')) \end{array}$$

This says essentially that t and t' can be decomposed as $a \otimes b$ and $a' \otimes b'$ respectively. However, the choice of the witnesses a, a', b and b' can depend on the initial states. The witnesses are not uniform across all states. Note that a and b depend only on x and y whereas a' and b' depend only on x' and y'. We can make this explicit by writing a_{xy}, $a'_{x'y'}$ etc. instead of simple variables a, a'.

The projections $(\pi_1)_{X,Y} : X \star Y \to X$ and $(\pi_2)_{X,Y} : X \star Y \to Y$ are defined similar to those of independent products.

Terminal object

The terminal object in $\mathbf{RTM}$, representing the "empty store," is $\mathbf{1} = (\mathbf{1}, \mathbf{0_1})$, where $\mathbf{0_1} = \{j_{\mathbf{1}}, \overline{\perp}\}$. The unique morphism $!_X : X \to \mathbf{1}$ is $!_X = (!_{\mathcal{Q}_X}, \tau_{!_X})$ where $\tau_{!_X}$ sends $j_{\mathbf{1}}$ to $j_{\mathcal{Q}_X}$ and $\overline{\perp}$ to $\overline{\perp}$. These morphisms are parametric in X, i.e., if $R : X \leftrightarrow X'$, then $!_X \left[R \to I_{\mathbf{1}}\right] !_{X'}$.

The terminal object is the unit for both forms of products: $A \otimes \mathbf{1} \cong A$ and $A \star \mathbf{1} \cong A$.

VI. Semantics

The reflexive graph $\mathbf{CPO}$ consists of directed-complete partial orders as objects, continuous functions as morphisms, and directed-complete relations as edges. It is a parametricity graph. The weakest pre-edge $[f, f']R$ is the pre-image $\{\, (x, x') \mid f(x) \left[R\right] f'(x') \,\}$ which is easily seen to be a directed-complete relation. Note that the "graph" of a morphism $f : A \to A'$ (in the formal sense) is $[f, \mathrm{id}_{A'}]I_{A'}$, which is nothing but the graph of the continuous function f.

The reflexive graph $\mathbf{CPO}_\perp$ consists of pointed cpo's (cpo's with least elements) as objects, strict continuous functions as morphisms and complete relations that relate least elements as edges. (A "complete" relation is a directed-complete relation that also relates the least elements.) It is also a parametricity graph.

We will be interested in PG-functors $F : \mathbf{RTM}^{\mathrm{op}} \to \mathbf{CPO}$ that factor through the embedding $J : \mathbf{CPO}_\perp \hookrightarrow \mathbf{CPO}$. That means that $F(X)$ is a pointed cpo for each rtm X, $F(f)$ is a strict continuous function for each morphism f of rtm's and $F(R)$ is a pointed complete relation for each relation R of rtm's. Such functors form a category $\mathcal{C}(\mathbf{RTM})$ with parametric transformations as morphisms.

Theorem 3 If $\mathbf{C}$ is an op-parametricity graph, let $\mathcal{C}(\mathbf{C})$ denote the category of PG-functors $\mathbf{C}^{\mathrm{op}} \to \mathbf{CPO}$ that factor through the embedding $J : \mathbf{CPO}_\perp \hookrightarrow \mathbf{CPO}$. Then $\mathcal{C}(\mathbf{C})$ is cartesian closed.

Products are given pointwise: $(F \times G)(X) = F(X) \times G(X)$ and $(F \times G)(R) = F(R) \times G(R)$. Exponents are given as in presheaf categories: $(F \Rightarrow G)(X) = \forall_{h:Z \to X}[F(Z) \to G(Z)]$, where $\forall$ denotes the "parametric limit" (in $\mathbf{CPO}$) indexed by morphisms h leading to X [9]. Explicitly, the parametric limit consists of families of the form $\{t_h \in [F(Z) \to G(Z)]\}_{h:Z \to X}$ that are parametric in the sense that $h \left[S \to I_X\right] h' \Rightarrow t_h \left[F(S) \to G(S)\right] t_{h'}$. Since F and G are PG-functors, such families are automatically natural [9]. It can be verified that it is a pointed cpo under the component-wise ordering. The relation $\forall_{S \to R}[F(S) \to G(S)]$ relates two families $\{t_h\}_{h:Z \to X}$ and $\{t'_{h'}\}_{h':Z' \to X'}$ iff, for all relations $S : Z \leftrightarrow Z'$ and all h, h' of appropriate types:

$$h \left[S \to R\right] h' \Longrightarrow t_h \left[F(S) \to G(S)\right] t'_{h'}$$

■

$\mathcal{C}(\mathbf{C})$ also extends to a parametricity graph, which is a fact used in interpreting *polymorphic* Algol-like languages [10]. However, we will not need this extension for the present purposes.

All types of IA+ can be interpreted in $\mathcal{C}(\mathbf{RTM})$. The interpretation is shown in Fig. 2. To avoid excessive bracketing, we use names like Comm etc. for semantic functors, instead of the usual notation of semantic brackets ($[\![\mathbf{comm}]\!]$ etc.) We also identify the names of data types δ with the sets of values denoted by them. For brevity, we omit the $\mathbf{val}[\delta]$ types and record types, which can be handled in a straightforward manner. Note that variables are interpreted as objects with *get* and *put* methods as described in Sec. III, except that we are now representing it as a pair of methods instead of a record of methods. The product and exponential constructions are from Theorem 3. The interpretation of classes (cls F) involves a hidden world for the data representation (an rtm) along with an initial state in that world and an implementation of the method suite in the world. Recall that we are only treating "constant classes" with no free identifiers. Such a class does not depend on the non-local store, and therefore (cls F)(R) is the identity relation.

The notation $\exists_Z\, T(Z)$ stands for the "parametric colimit," which is a quotient of $\coprod_Z T(Z)$ under the transitive closure of the *similarity* relation "$\sim$", which is defined by the rule:

$$S : Z \leftrightarrow Z' \wedge a \left[T(S)\right] a' \Rightarrow \langle Z, a\rangle \sim \langle Z', a'\rangle$$

The equivalence class of $\langle Z, a\rangle$ under $\sim^*$ is denoted by $\langle\!\langle Z, a\rangle\!\rangle$ and we call such an entity a "package." The relation $\exists_S\, T(S)$ relates two packages $\langle\!\langle Z, a\rangle\!\rangle$ and $\langle\!\langle Z', a'\rangle\!\rangle$ iff there exists a

$$
\begin{array}{rcl}
\textsc{Comm}(X) & = & \mathcal{T}_X \\
\textsc{Exp}_\delta(X) & = & [\mathcal{Q}_X \rightharpoonup \delta] \\
\textsc{Var}_\delta(X) & = & \textsc{Exp}_\delta(X) \times [\delta \to \textsc{Comm}(X)] \\
(F \times G)(X) & = & F(X) \times G(X) \\
(F \Rightarrow G)(X) & = & \forall_{h:Z\to X}\, [F(Z) \to G(Z)] \\
(\textsc{cls}\ F)(X) & = & \exists_Z\, (\mathcal{Q}_Z)_\perp \times F(Z)
\end{array}
\qquad
\begin{array}{rcl}
\textsc{Comm}(R) & = & R_T \\
\textsc{Exp}_\delta(R) & = & [R_Q \rightharpoonup \Delta_\delta] \\
\textsc{Var}_\delta(R) & = & \textsc{Exp}_\delta(R) \times [\Delta_\delta \to \textsc{Comm}(R)] \\
(F \times G)(R) & = & F(R) \times G(R) \\
(F \Rightarrow G)(R) & = & \forall_{S\to R}\, [F(S) \to G(S)] \\
(\textsc{cls}\ F)(R) & = & I_{(\textsc{cls}\ F)(X)}
\end{array}
$$

Fig. 2. Interpretation of IA+ types

relation $S\colon Z \to Z'$ such that $a \left[T(S)\right] a'$. These notions are discussed in detail in our prior work [25, 28].[6]

To complete the definition, we need to specify the action of the functors on morphisms and show that they constitute PG-functors. The action on morphisms can be uniquely reconstructed from the action on edges because, if F is a PG-functor, then $F(\langle f\rangle) = \langle F(f)\rangle$, i.e., $F(\langle f\rangle)$ is the graph of a strict-continuous function. There is evidently at most one such function. We exhibit these functions for the functors involved in the interpretation of IA+:

- $\textsc{Comm}(f) = \tau_f$, which is strict and continuous by definition.
- $\textsc{Exp}_\delta(f) = [\phi_f \rightharpoonup \mathrm{id}_\delta]$ sends an expression valuation $e \in \textsc{Exp}_\delta(X)$ to $e \circ \phi_f \in \textsc{Exp}_\delta(Y)$, which is evidently strict and continuous.
- $(F \times G)(f) = F(f) \times G(f)$, which preserves strictness and continuity.
- $(F \Rightarrow G)(f\colon X' \to X)$ sends a family $\{t_h \in [F(Z) \to G(Z)]\}_{h:Z\to X}$ to the corresponding family $\{t_{(h';f)}\}_{h':Z\to X'}$, which is evidently strict and continuous.
- $(\textsc{cls}\ F)(f\colon X' \to X)$ is just the identity morphism $\mathrm{id}_{(\textsc{cls}\ F)(X)}$.

Using these functor actions, we can upgrade any value d of type θ at world X to a future world Y. When the morphism $f : Y \to X$ is clear from the context, we often use the shorthand notation $d{\uparrow}_X^Y \triangleq [\![\theta]\!](f)(d)$ to denote such upgrading.

Interpretation of terms

The meaning of a term M with typing:

$$x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta$$

is a parametric transformation of type

$$[\![M]\!] : (\textstyle\prod_{x_i} [\![\theta_i]\!]) \to [\![\theta]\!]$$

This means that, for each world (rtm) X, $[\![M]\!]_X$ is a continuous function of type $(\prod_{x_i} [\![\theta_i]\!](X)) \to [\![\theta]\!](X)$ such that all relations are preserved, i.e., for any relation $R : X \leftrightarrow X'$, we have $[\![M]\!]_X \left[(\prod_{x_i} [\![\theta_i]\!](R)) \to [\![\theta]\!](R)\right] [\![M]\!]_{X'}$. To the extent that IA+ is a simply typed lambda calculus, this is standard [9, 19]. We show the basic constructs:

$$
\begin{array}{l}
[\![x]\!]_X(u) = u(x) \\
[\![\lambda x\colon \theta.\, M]\!]_X(u) = \\
\qquad \Lambda h\colon Z \to X.\, \lambda d\colon [\![\theta]\!](Z).\, [\![M]\!]_Z(u{\uparrow}_X^Z[x \mapsto d]) \\
[\![M\,N]\!]_X(u) = [\![M]\!]_X(u)[\mathrm{id}_X : X \to X]([\![N]\!]_X(u))
\end{array}
$$

The parameter u may be thought of as an "environment" that provides values for the free identifiers, specifically in the given world X. The meaning of a lambda abstraction of type $\theta \to \theta'$ is in $([\![\theta]\!] \Rightarrow \theta')(X)$, which consists of families of the form $\{t_h\}_{h:Z\to X}$. Here, we are using notation "$\Lambda h : Z \to X$" borrowed from the polymorphic lambda calculus to express the h parameter. Note that the body of the abstraction is interpreted in the future world Z and the environment u is upgraded to this world. Parametricity in Z is crucial for capturing the fact that $[\![M]\!]_Z$ does not directly access any information of the future world. In the interpretation of function application terms, we are again using the polymorphic lambda calculus notation to pass in the h parameter, viz., $\mathrm{id}_X : X \to X$.

The interpretation of class definitions is given by:

$$
\begin{array}{c}
[\![\textbf{class} : \theta\ \textbf{local}\ C\ x\ \textbf{init}\ A\ \textbf{meth}\ M]\!]_X(u) = \\
\langle\!\langle Z,\ ([\![A]\!]_Z(u_0))^*(z_0),\ [\![M]\!]_Z(u_0)\rangle\!\rangle \\
\text{where } \langle\!\langle Z, z_0, m\rangle\!\rangle = [\![C]\!]_X(u) \\
u_0 = \{x \mapsto m\}
\end{array}
$$

This says that the package for the class C is opened and a new package for the class term is created using it. We are depending on the fact that the class definition is a closed term. So, the only free identifier in A and M is x.

The interpretation of the $\mathbf{new}$ construct for creating class instances is:

$$
\begin{array}{l}
[\![\mathbf{new}\ C\ P]\!]_X(u) = \\
\qquad (\lambda s.\, [s, z_0]_\perp) \cdot [\![P]\!]_X(u)[(\pi_1)_{X,Z}](m{\uparrow}_Z^{X\star Z}) \cdot (\lambda(s,z).\, s) \\
\qquad\quad \text{where } \langle\!\langle Z,\ z_0, m\rangle\!\rangle = [\![C]\!]_X(u)
\end{array}
$$

The interpretation extends the current world X to $X \star Z$, where Z is a store for the internal state of the class, and executes the body of the $\mathbf{new}$ operator (P) in the extended store. This execution is bracketed with an allocation and deallocation of

[6] We are glossing over some detail here because constructing colimits of cpo's is a nontrivial exercise [14]. We prefer the alternative solution outlined in [25], where CPO's are coupled with partial equivalence relations to define more manageable colimits.

equal : $\textsc{Exp}_\delta \times \textsc{Exp}_\delta \to \textsc{Exp}_{Bool}$	$\text{equal}(e_1, e_2) = \lambda s.\, (\lambda(d_1, d_2).\, d_1 = d_2)^*\, [e_1(s), e_2(s)]_\perp$
cond^E : $\textsc{Exp}_{Bool} \times \textsc{Exp}_\delta \times \textsc{Exp}_\delta \to \textsc{Exp}_\delta$	$\text{cond}^E_X(e, e_1, e_2) = \lambda s.\, (\lambda v.\, v \to e_1(s);\ e_2(s))^*(e(s))$
skip : $1 \to \textsc{Comm}$	$\text{skip}_X(*) = \text{null}_X$
seq : $\textsc{Comm} \times \textsc{Comm} \to \textsc{Comm}$	$\text{seq}_X(a, b) = a \cdot b$
cond^C : $\textsc{Exp}_{Bool} \times \textsc{Comm} \times \textsc{Comm} \to \textsc{Comm}$	$\text{cond}^C_X(e, a, b) = \text{read}_X\ \lambda s.\, (\lambda v.\, v \to a;\ b)^*(e(s))$
deref : $\textsc{Var}_\delta \to \textsc{Exp}_\delta$	$\text{deref}_X(e, a) = e$
assign : $\textsc{Var}_\delta \times \textsc{Exp}_\delta \to \textsc{Comm}$	$\text{assign}_X((d, a), e) = \text{read}_X\ \lambda s.\, a^*(e(s))$
$\mathbf{var}[\delta] : 1 \to \textsc{cls}\ \textsc{Var}_\delta$	$\mathbf{var}[\delta]_X(*) = \langle\!\langle V, \mathit{init}_\delta, \text{mkvar} \rangle\!\rangle$ where $V = (\delta, T(\delta))$ mkvar $= (\lambda n.\, n,\ \lambda k.\, \lambda n.\, k)$
newvar : $(\textsc{Var}_\delta \Rightarrow \textsc{Comm}) \to \textsc{Comm}$	$\text{newvar}_X(p) = (\lambda s.\, (s, \mathit{init}_\delta)) \cdot p[\pi_1](\text{mkvar}\uparrow^{X \star V}_V) \cdot (\lambda(s, n).\, s)$

Fig. 3. Primitive operators of IA+

the class instance, so that the overall command is still in the world X.

The interpretations of the primitives (constants) of IA+ is shown in Fig. 3. (Recall that the notation f^* extends a function $f : A \rightharpoonup B$ to the type $A_\perp \to B_\perp$.)

The primitive $\mathbf{var}[\delta]$ requires some explanation. Variables are treated in Idealized Algol as "objects" with methods for reading and writing their values (of types $\textsc{Exp}_\delta$ and $\delta \to \textsc{Comm}$ respectively). We use the shorthand $\textsc{Var}_\delta = \textsc{Exp}_\delta \times (\delta \to \textsc{Comm})$ for the type of variables. In the world $V = (\delta, T(\delta))$, we can define a value *mkvar* that uses the states of the world V to construct the two methods. The constant init_δ represents some global value that is presumed to be used as the initial value for variables of type δ.

To give additional insight, we also show a primitive called newvar, which is nothing but $\mathbf{new\ var}[\delta]$. Given any world X, we have the expanded world $X \star V$ with projections $\pi_1 : X \star V \to X$ and $\pi_2 : X \star V \to V$ and $\text{mkvar}\uparrow^{X \star V}_V = \textsc{Var}_\delta(\pi_2)(\text{mkvar}) \in \textsc{Var}_\delta(X \star V)$. This variable object is provided as the argument to p. The remaining steps of newvar_X are the allocation and the deallocation of the local variable.

Lemma 4 All the combinators of Idealized Algol are parametric transformations.

Theorem 5 The meaning of every IA+ term $x_1 : \theta_1, \ldots, x_n : \mathit{theta}_n \vdash M : \theta$ is a parametric transformation of type $(\prod_{x_i} [\![\theta_i]\!]) \to [\![\theta]\!]$.

This completes the semantic definition of IA+.

Example equivalences

Example 6 We can define two classes for counter objects as follows:[7]

$$\begin{aligned} \mathit{counter}_1 = \mathbf{class} : \mathbf{exp}[\mathbf{int}] \times \mathbf{comm} \\ \mathbf{local\ var}[\mathbf{int}]\ x; \\ \mathbf{init}\ x := 0; \\ \mathbf{meth}\ (\text{deref}\ x,\ x := x + 1) \\ \mathit{counter}_2 = \mathbf{class} : \mathbf{exp}[\mathbf{int}] \times \mathbf{comm} \\ \mathbf{local\ var}[\mathbf{int}]\ x; \\ \mathbf{init}\ x := 0; \\ \mathbf{meth}\ (-(\text{deref}\ x),\ x := x - 1) \end{aligned}$$

Their meanings should be semantic values of type:

$$\exists_Z\, (\mathcal{Q}_Z)_\perp \times (\textsc{Exp}_{Int} \times \textsc{Comm})(Z)$$

The meaning of the class $\mathit{counter}_1$ is as follows:

- The store Z_1 for the object is given by

$$\begin{aligned} \mathcal{Q}_{Z_1} &= \mathit{Int} \\ \mathcal{T}_{Z_1} &= \text{read-closure of } \{\overline{\perp}\} \cup \{\, \text{inc}(k) \mid k \geq 0 \,\} \\ &\qquad \text{where inc}(k) = \lambda n.\, n + k \end{aligned}$$

  Note that $\mathcal{T}_{Z_1}$ is a monoid with the unit element inc(0).
- The initial value is 0.
- The method suite in $(\textsc{Exp}_{Int} \times \textsc{Comm})(Z_1)$ is the pair:

$$\mathit{meth}_1 = ((\lambda n.\, n),\ \text{inc}(1))$$

The meaning of the class $\mathit{counter}_2$ is similar:

- The future world is $Z_2 = X \star K_2$ where K_2 is given by

$$\begin{aligned} \mathcal{Q}_{Z_2} &= \mathit{Int} \\ \mathcal{T}_{Z_2} &= \text{read-closure of } \{\overline{\perp}\} \cup \{\, \text{dec}(k) \mid k \geq 0 \,\} \\ &\qquad \text{where dec}(k) = \lambda n.\, n - k \end{aligned}$$

[7] Since the class is defined using a variable object, the semantic definition states the meaning in terms of the world V for the internal state of the counter, which includes the full transformation monoid $(\delta, T(\delta))$. However, the meaning of the class is an abstract "package," unique upto behavioral equivalence. So, we can cut down the transformation component of the world to just those transformations directly used in the class via behavioral equivalence.

- The initial value is 0.
- The method suite in $(\textsc{Exp}_{Int} \times \textsc{Comm})(Z_2)$ is the pair:

$$\mathit{meth}_2 = ((\lambda n.\, -n),\ \mathrm{dec}(1))$$

To demonstrate that the two classes are equal in the parametric colimit, we can exhibit a relation $R : Z_1 \leftrightarrow Z_2$ that is preserved by the initialization and the method suite. The relation is $S : Z_1 \leftrightarrow Z_2$, given by:

$$\begin{array}{rcl} S_Q & = & \{\, (n, -n) \mid n \geq 0 \,\} \\ S_T & = & \{(\overline{\bot}, \overline{\bot})\} \cup \{\, (\mathrm{inc}(k),\ \mathrm{dec}(k)) \mid k \geq 0 \,\} \end{array}$$

The preservation properties to be verified are:

$$\begin{array}{l} 0 \left[(S_Q)_\bot\right] 0 \\ \mathit{meth}_1 \left[(\textsc{Exp}_{Int} \times \textsc{Comm})(S)\right] \mathit{meth}_2 \end{array}$$

It is easy to verify them once we note that $(\textsc{Exp}_{Int} \times \textsc{Comm})(S) = [S_Q \rightharpoonup \Delta_{Int}] \times S_T$. ■

Example 7 (Pitts and Stark "awkward" example) Consider the following classes:

$$\begin{array}{l} C_1 = \mathbf{class} : \mathbf{comm} \to \mathbf{comm} \\ \qquad \mathbf{local\ var}[\mathbf{int}]\ x; \\ \qquad \mathbf{init}\ x := 0; \\ \qquad \mathbf{meth}\ \lambda c.\, x := 1; c; \mathit{test}(x = 1) \\ C_2 = \mathbf{class} : \mathbf{comm} \to \mathbf{comm} \\ \qquad \mathbf{local\ var}[\mathbf{int}]\ x; \\ \qquad \mathbf{init}\ x := 0; \\ \qquad \mathbf{meth}\ \lambda c.\, c \end{array}$$

where $\mathit{test}(b) = \mathbf{if}\ b\ \mathbf{then\ skip\ else\ diverge}$.

A relation S between the internal states of the classes C_1 and C_2 has two components, a relation S_Q between their state sets and a relation S_T between their state transformations. The transformation component S_T relates the transformations null and put(1) of C_1 to the null transformation of C_2. Since the c arguments to the methods are assumed to be related by S_T, we can conclude that the call to c in C_1 executes some combination of null and put(1) actions, with the result that x is 1 after the call.

We show the detailed proof. The meanings of the classes should be semantic values of type:

$$\exists_Z \left((\mathcal{Q}_Z)_\bot \times \forall_{g:Y \to Z}\, \textsc{Comm}(Y) \to \textsc{Comm}(Y)\right)$$

The meaning of the class C_1 is as follows:

$$\begin{array}{l} \mathcal{Q}_{Z_1} = \mathit{Int} \\ \mathcal{T}_{Z_1} = \text{read-closure of } \{\overline{\bot}, \mathrm{null}_{Z_2}, \mathrm{put}(1)\} \\ \mathit{init}_1 = 0 \\ \mathit{meth}_1 = \Lambda g\colon Y \to Z_1.\, \lambda c\colon \textsc{Comm}(Y). \\ \qquad\qquad \mathrm{put}(1)\uparrow^{Y}_{Z_1} \cdot c \cdot \mathrm{check}(1)\uparrow^{Y}_{Z_1} \end{array}$$

where $\mathrm{put}(k) = \lambda n.\, k$ and $\mathrm{check}(k) = \mathrm{read}\ \lambda n.\, n = k \to \mathrm{null};\ \overline{\bot}$.

The meaning of the class C_2 is similar:

$$\begin{array}{l} \mathcal{Q}_{Z_2} = \mathit{Int} \\ \mathcal{T}_{Z_2} = \text{read-closure of } \{\overline{\bot}, \mathrm{null}_{Z_2}\} \\ \mathit{init}_2 = 0 \\ \mathit{meth}_2 = \Lambda g\colon Y \to Z_2.\, \lambda c\colon \textsc{Comm}(Y).\, c \end{array}$$

To demonstrate that the two classes are equal, we exhibit a relation $S : Z_1 \leftrightarrow Z_2$ given by:

$$\begin{array}{rcl} S_Q & = & \{\, (n, 0) \mid n \geq 0 \,\} \\ S_T & = & \{(\overline{\bot}, \overline{\bot}),\ (\mathrm{null}_{Z_1}, \mathrm{null}_{Z_2}),\ (\mathrm{put}(1), \mathrm{null}_{Z_2})\} \end{array}$$

The preservation properties to be verified are:

$$\begin{array}{l} \mathit{init}_1 \left[(S_Q)_\bot\right] \mathit{init}_2 \\ \mathit{meth}_1 \left[(\textsc{Comm} \Rightarrow \textsc{Comm})(S)\right] \mathit{meth}_2 \end{array}$$

Note that $(\textsc{Comm} \Rightarrow \textsc{Comm})S = \forall_{R \to S}\, \textsc{Comm}(R) \to \textsc{Comm}(R) = \forall_{R \to S}\, R_T \to R_T$. So, the relationship to be proved between the two method suites is:

$$\begin{array}{l} \forall g_1\colon Y_1 \to Z_1.\, \forall g_2\colon Y_2 \to Z_2.\, g_1 \left[R \to S\right] g_2 \Longrightarrow \\ \quad \forall c_1, c_2.\, c_1 \left[R_T\right] c_2 \Longrightarrow \\ \qquad \mathit{meth}_1[g_1](c_1) \left[R_T\right] \mathit{meth}_2[g_2](c_2) \end{array}$$

Since put(1) $\left[S_T\right]$ null, we have $\mathrm{put}(1)\uparrow^{Y_1}_{Z_1} \left[R_T\right] \mathrm{null}\uparrow^{Y_2}_{Z_2}$. Since $c_1 \left[R_T\right] c_2$ by assumption, the state in Z_1 (the value of x) is 1, as argued above. Therefore check(1) has the effect of null_{Z_1}. Hence, we have the required property. ■

Example 8 (Dreyer, Neis and Birkedal) Consider the following classes:

$$\begin{array}{l} C_1 = \mathbf{class} : \mathbf{comm} \to \mathbf{comm} \\ \qquad \mathbf{local\ var}[\mathbf{int}]\ x; \\ \qquad \mathbf{init}\ x := 0; \\ \qquad \mathbf{meth}\ \lambda c.\, x := 0; c; x := 1; c; \mathit{test}(x = 1) \\ C_2 = \mathbf{class} : \mathbf{comm} \to \mathbf{comm} \\ \qquad \mathbf{local\ var}[\mathbf{int}]\ x; \\ \qquad \mathbf{init}\ x := 0; \\ \qquad \mathbf{meth}\ \lambda c.\, c; c \end{array}$$

where $\mathit{test}(b) = \mathbf{if}\ b\ \mathbf{then\ skip\ else\ diverge}$.

This example is similar to the "awkward" example, except that we have two calls to c in the method of C_1, interspersed by different assignments to x. The differences from the above example are as follows:

$$\begin{array}{rcl} \mathcal{T}_{Z_1} & = & \text{read-closure of } \{\overline{\bot}, \mathrm{null}_{Z_1}, \mathrm{put}(0), \mathrm{put}(1)\} \\ \mathit{meth}_1 & = & \Lambda g\colon Y \to Z_1.\, \lambda c\colon \textsc{Comm}(Y). \\ & & \quad \mathrm{put}(0)\uparrow^{Y}_{Z_1} \cdot c \cdot \mathrm{put}(1)\uparrow^{Y}_{Z_1} \cdot c \cdot \mathrm{check}(1)\uparrow^{Y}_{Z_1} \\ \mathcal{T}_{Z_2} & = & \text{read-closure of } \{\overline{\bot}, \mathrm{null}_{Z_2}\} \\ \mathit{meth}_2 & = & \Lambda g\colon Y \to Z_2.\, \lambda c\colon \textsc{Comm}(Y).\, c \cdot c \\ S_T & = & \{(\overline{\bot}, \overline{\bot}),\ (\mathrm{null}_{Z_1}, \mathrm{null}_{Z_2}),\ (\mathrm{put}(1), \mathrm{null}_{Z_2})\} \end{array}$$

It is worth noting that the relation $S_T : \mathcal{T}_{Z_1} \leftrightarrow \mathcal{T}_{Z_2}$ is the *same* as that in the awkward example.

We verify the simulation property

$$\mathit{meth}_1 \left[(\textsc{Comm} \Rightarrow \textsc{Comm})(S)\right] \mathit{meth}_2$$

as in the previous example, which involves the condition:

$$\forall g_1 \colon Y_1 \to Z_1 . \forall g_2 \colon Y_2 \to Z_2 .\, g_1 \left[R \to S\right] g_2 \Longrightarrow \\ \forall c_1, c_2 .\, c_1 \left[R_T\right] c_2 \Longrightarrow \\ \mathit{meth}_1[g_1](c_1) \left[R_T\right] \mathit{meth}_2[g_2](c_2)$$

We first argue that $\mathit{meth}_1[g_1](c_1)$ and $\mathit{meth}_2[g_2](c_2)$ are related by $R_Q \rightharpoonup R_Q$. Starting from related initial states n and 0, the first action in meth_1 is $\mathrm{put}(0)$, which changes the local state to 0. Calling c has the effect of either null_{Z_1} or $\mathrm{put}(1)$ on x. So, x is either 0 or 1, both of which are related to 0 by R_Q. The next action $\mathrm{put}(1)$ overrides the previous effect and changes the local state to 1. The second call to c again has the effect of either null_{Z_1} or $\mathrm{put}(1)$, with the result that the local state continues to be 1 and, so, $\mathrm{check}(1)$ succeeds. Thus, the overall effect of meth_1 is to set the local state to 1, i.e., a $\mathrm{put}(1)$ action, and two calls to c for the effects on the non-local state. This is related to $c \cdot c$ in meth_2 by the R_T relation.

Dreyer et al. [6] characterize actions as $\mathrm{put}(0)$ in meth_1 as "private transitions" because they are not visible at the end of method calls. Note that no special treatment is needed in the semantics to capture such private transitions. Essentially, the private transitions are handled by R_Q, the state components of the rtm-relations, whereas the public transitions are handled by R_T, the transformation components of the relations. ■

VII. Conclusion

We have outlined a new denotational semantic model for class-based Algol-like languages, which combines the advantages of the existing models. Similar to the state-based models, it is able to represent the effect of operations as state transformations. At the same time, it also represents stores as rudimentary form of objects, whose state changes are treated from the outside in a modular fashion. Further, this modeling allows one to prove observational equivalences of programs that were not possible in the previous models. This work complements that of Ahmed, Dreyer and colleagues [3, 6] who use an *operational approach* to develop similar reasoning principles.

In principle, this work could have been done any time after 1983, because Reynolds used a similar framework for his semantics in [30] and formulated relational parametricity in [32]. We can only speculate why it wasn't done. The alternative model invented by Oles [21] was considered equivalent to the Reynolds's model and it appeared to be simpler as well as more general. However, sharp differences between the two models become visible as soon as relational parametricity is considered. This fact was perhaps not appreciated in the intervening years.

In terms of further work to be carried out, we have not addressed the issues of dynamic storage (pointers) but we expect that the prior work in parametricity semantics [29] will be applicable. We have not considered higher-order store, i.e., storing procedures in variables. This problem is known to be hard in the framework of functor category models and it may take some time to get resolved. More exciting work awaits to be done in applying these ideas to study program reasoning, including Specification Logic [31, 33], Separation Logic, Rely-guarantee and Deny-guarantee reasoning techniques [5, 34].

References

[1] S. Abramsky and G. McCusker. Linearity, sharing and state. In *Algol-like Languages* O'Hearn and Tennent [20], chapter 20.

[2] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS 1998*, pages 334–344, 1998.

[3] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Thirty Sixth Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 2009.

[4] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *European Conf. on Object-Oriented Programming - ECOOP*, pages 504–528, Berlin, 2010. Springer-Verlag.

[5] M. Dodds, M. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *European Symposium on Programming*, pages 363–377, Berlin, 2009. Springer-Verlag.

[6] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *IFCP*, 2010.

[7] D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. A relational modal logic for higher-order stateful ADTs. In *Thirty Seventh Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 2010.

[8] B. P. Dunphy. *Parametricity as a Notion of Uniformity in Reflexive Graphs*. PhD thesis, University of Illinois, Dep. of Mathematics, 2002. available electronically from http://www.cs.bham.ac.uk/˜udr.

[9] B. P. Dunphy and U. S. Reddy. Parametric limits. In *Proc. 19th Ann. IEEE Symp. on Logic in Comp. Sci.*, pages 242–253. IEEE, July 2004.

[10] B. P. Dunphy and U. S. Reddy. Semantics of parametric polymorphism in imperative programming languages. Electronic manuscript, University of Birmingham, Oct 2010.

[11] S. Eilenberg. *Automata, Languages, and Machines*. Academic Press, 1974. (Volumes A and B).

[12] M. P. Fiore, A. Jung, E. Moggi, P. W. O'Hearn, J. Riecke, G. Rosolini, and I. Stark. Domains and denotational semantics: History, accomplishments and open problems. *Bulletin of the European Assoc. for Theor. Computer Science*, 59:227–256, 1996.

[13] W. M. L. Holcombe. *Algebraic Automata Theory*. Cambridge Studies in Advanced Mathematics. Cambridge Univ. Press, Cambridge, 1982.

[14] A. Jung. Colimits in DCPO. Unpublished Manuscript, 1990.

[15] 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 [20]).

[16] P. W. O'Hearn and U. S. Reddy. Objects, interference and

the Yoneda embedding. *Theoretical Computer Science*, 228(1):211–252, 1999.

[17] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, Jan 2000.

[18] P. W. O'Hearn and J. G. Riecke. Kripke logical relations and PCF. *Inf. Comput.*, 120(1):107–116, 1995.

[19] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995. (Reprinted as Chapter 16 of [20]).

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

[21] F. J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573. Cambridge Univ. Press, 1985.

[22] A. M. Pitts and I. D. B. Stark. Operational reasonign for functions with local state. In A. M. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–274. Cambridge Univ. Press, Cambridge, 1998.

[23] 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 [20]).

[24] U. S. Reddy. When parametricity implies naturality. Electronic manuscript, University of Birmingham, http://www.cs.bham.ac.uk/˜udr/, July 1997.

[25] U. S. Reddy. Objects and classes in Algol-like languages. In *FOOL 5: Fifth Intern. Workshop on Foundations of Object-oriented Languages*. electronic proceedings at http://pauillac.inria.fr/˜remy/ fool/proceedings.html, Jan 1998.

[26] U. S. Reddy. Parametricity and naturality in the semantics of Algol-like languages. Electronic manuscript, University of Birmingham, http://www.cs.bham.ac.uk/˜udr/, Dec 1998.

[27] U. S. Reddy. Parametricity and naturality in the semantics of Algol. Talk at the *Intern. Workshop on Math. Found. of Program. Semantics*, London, May 1998.

[28] U. S. Reddy. Objects and classes in Algol-like languages. *Information and Computation*, 172:63–97, 2002.

[29] U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1-3):129–160, Mar 2004.

[30] 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 [20]).

[31] 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 [20]).

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

[33] R. D. Tennent. Semantical analysis of specification logic. *Inf. Comput.*, 85(2):135–162, 1990. (Reprinted as Chapter 13 of [20]).

[34] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and Separation Logic. In *CONCUR 2007*, volume 4703 of *LNCS*, pages 256–271, 2007.

APPENDIX

Proof of Lemma 1

If $a \in \mathcal{T}_X$ and $a' \sqsubseteq a$, there is a function $p' : \mathcal{Q}_X \to \mathcal{T}_X$ given by $p' = \lambda x.\ a'(x) \neq \bot \to a;\ \overline{\bot}$. It is easy to see that $a' = \mathrm{read}_X(p')$.

Assume $a\ [R]\ b$, $(a', b') \sqsubseteq (a, b)$ and $a'\ [R_Q \rightharpoonup \mathbb{T}]\ b'$. The last of these means that, for all x, y such that $x\ [R_Q]\ y$, we have $(a'(x) = \bot \wedge b'(y) = \bot) \vee (a'(x) \neq \bot \wedge b'(y) \neq \bot)$. In other words, $a'(x) = \bot \iff b'(y) = \bot$. Given the assumptions, we can construct $p' : \mathcal{Q}_X \to \mathcal{T}_X$ and $q' : \mathcal{Q}_Y \to \mathcal{T}_Y$ as above, giving $a' = \mathrm{read}_X(p')$ and $b' = \mathrm{read}_Y(q')$. If $x\ [R_Q]\ y$ then $a'(x) = \bot \iff b'(y) = \bot$, which implies $p'(x)\ [R]\ q'(y)$. Hence, $p'\ [R_Q \to R_T]\ q'$ and $a'\ [R]\ b'$. ■

Proof of Lemma 4

We show selected cases. For the assignment operation, let $(d, a)\ [\textsc{Var}_\delta(R)]\ (d', a')$ and $e\ [\textsc{Exp}_\delta(R)]\ e'$. Then

$$(\lambda s.\, a^*(e(s)))\ [R_Q \to R_T]\ (\lambda s'.\, a'^*(e'(s')))$$
$$(\mathrm{read}_X\ \lambda s.\, a^*(e(s)))\ [R_T]\ (\mathrm{read}_{X'}\ \lambda s'.\, a'^*(e'(s')))$$

The second step follows from the fact that the relations are compatible with the diagonal operation.

Consider the newvar combinator. Let $R : X \leftrightarrow X'$ be a relation of rtm's and assume $p\ [(\textsc{Var}_\delta \Rightarrow \textsc{Comm})(R)]\ p'$.

1) The relation $(\textsc{Var} \Rightarrow \textsc{Comm})(R)$ is $\forall_{S \to R}\, \textsc{Var}(S) \to \textsc{Comm}(S)$. In the particular case used in the combinator, S is instantiated to $R \star I_V : X \star V \leftrightarrow X' \star V$. So, we obtain $p[(\pi_1)_{X,V}]\ \left[\textsc{Var}(R \star I_V) \to \textsc{Comm}(R \star I_V)\right]\ p[(\pi_1)_{X',V}]$.
2) We argue that $\mathrm{mkvar}{\uparrow}_V^X$ and $\mathrm{mkvar}{\uparrow}_V^{X'}$ are related by $\textsc{Var}_\delta(R \star I_V)$. Firstly, $\lambda(s, n).\, n$ and $\lambda(s', n').\, n'$ are related by $\textsc{Exp}_\delta(R \star I_V)$, i.e., $R_Q \times \Delta_\delta \rightharpoonup \Delta_\delta$. Secondly, $\lambda k.\, \lambda(s, n).\, (s, k)$ and $\lambda k'.\, \lambda(s', n').\, (s', k')$ are related by $\Delta_\delta \to \textsc{Comm}(R \star I_V)$, i.e., $\Delta_\delta \to R_T \star \Delta_{T(\delta)}$. Note that $\lambda(s, n).\, (s, k)$ can be expressed as $\mathrm{null}_X \otimes (\lambda n.\, k)$ in $\mathcal{T}_X \star T(\delta)$. null_X and $\mathrm{null}_{X'}$ are related by $\mathcal{T}_X$ and $\lambda n.\, k$ is related to itself by $\Delta_{T(\delta)}$.
3) Therefore, $p[(\pi_1)_{X,V}](\mathrm{mkvar}{\uparrow}_V^X)\ \left[\textsc{Comm}(R \star I_V)\right]\ p[(\pi_1)_{X',V}](\mathrm{mkvar}{\uparrow}_V^{X'})$.
4) The relation $\textsc{Comm}(R \star I_V)$ is $R_T \star (I_V)_T$ where $(I_V)_T = [\Delta_\delta \rightharpoonup \Delta_\delta]$. So, the instances $t = p[(\pi_1)](\mathrm{mkvar}{\uparrow}_V^X)$ and $t' = p'[(\pi_1)](\mathrm{mkvar}{\uparrow}_V^{X'})$ are related by $R_T \star [\Delta_\delta \rightharpoonup \Delta_\delta]$. So, for any $s \in \mathcal{Q}_X$, $s' \in \mathcal{Q}_{X'}$ and $n \in \delta$, there exist a_{sn}, $a'_{s'n}$, b_n such that $a_{sn}\ [R]\ a'_{s'n}$, $t(s, n) = (a_{sn} \otimes b_n)(s, n)$ and $t'(s', n) = (a'_{s'n} \otimes b_n)(s', n)$.
5) In particular, the above statement holds for $n = 0$. So, unless $b_0(0) = \bot$, $(t \cdot (\lambda(s, n).\, s))(s, 0) = a_{s0}(s)$ and $(t' \cdot (\lambda(s', n).\, s'))(s', 0) = a'_{s'0}(s')$. In other words, $\mathrm{newvar}_X(p)(s) = a_{s0}(s)$ and $\mathrm{newvar}_{X'}(p')(s') = a'_{s'0}(s')$. If, on the other hand, $b_0(0) = \bot$, both the functions evaluate to $\bot$. Hence, we can write $\mathrm{newvar}_X(p)$ as $\left(\mathrm{read}_X\ \lambda s.\, b_0(0) = \bot \to \bot;\ a_{s0}\right)$, and similarly for $\mathrm{newvar}_{X'}(p')$. These two transformations are related by R_T.

The case of cond^C illustrates how expression evaluations are embedded in commands. Again, let $R : X \leftrightarrow X'$ be a relation of rtm's and assume $e\ [\textsc{Exp}_\delta(R)]\ e'$, $a\ [\textsc{Comm}(R)]\ a'$ and $b\ [\textsc{Comm}(R)]\ b'$. To show that $\mathrm{cond}^C_X(e, a, b)\ [\textsc{Comm}(R)]\ \mathrm{cond}^C_{X'}(e', a', b')$, we need to show that $p = (\lambda s.(\lambda k.\, k \neq 0 \to a;\ b)^*(e(s)))$ and $p' = (\lambda s'.\, (\lambda k'.\, k' \neq 0 \to a';\ b')^*(e'(s')))$ are related by $R_Q \to R_T$. So, consider the action of the functions on states s and s' such that $s\ [R]\ s'$.

1) Since $\textsc{Exp}_\delta(R) = [R \rightharpoonup \Delta_\delta]$, we have $e(s)\ [\Delta_{\delta_\bot}]\ e'(s')$, i.e., $e(s) = e'(s')$.
2) If $e(s) = e'(s') = \bot$ then $p(s) = \bot_{\mathcal{T}_X}$ and $p'(s') = \bot_{\mathcal{T}_{X'}}$, which are related R_T since it is a pointed relation.
3) If $e(s) = e'(s') = 0$ then $p(s) = b$ and $p'(s') = b'$, which are related by R_T by assumption that b and b' are related by $\textsc{Comm}(R) = R_T$. The case of $e(s) = e'(s')$ being non-zero is similar.

All the other combinators can be similarly verified to be parametric. ■

Treatment of general classes

In the main body of the paper, we restricted attention to "constant classes" that have no free identifiers. Classes with free identifiers are quite useful, e.g., for defining nested classes. Here, we treat the general case. The interpretation of the general **cls** types is as follows:

$$\begin{aligned}(\textsc{cls}\ F)(X) &= \forall_{g:Y \to X}\, \exists_{h:Z \to Y} \\ &\qquad [\mathcal{Q}_Y \rightharpoonup \mathcal{Q}_Z] \times F(Z) \times [Q_Z \rightharpoonup Q_Y] \\ (\textsc{cls}\ F)(R) &= \forall_{P \to R}\, \exists_{S \to P} \\ &\qquad [P_Q \rightharpoonup S_Q] \times F(S) \times [S_Q \rightharpoonup P_Q]\end{aligned}$$

The meaning of a class at world X provides a way of creating instances at all future worlds Y, and such creation leads to a further future world Z. In addition to the method suite, of type $F(Z)$, we have allocation and deallocation operations, which are both irregular state transformations.

The notation $\exists_{h:Z \to X}\, T(Z)$ stands for an indexed "parametric colimit." It is a quotient of $\coprod_{h:Z \to X} T(Z)$ under the transitive closure of the "similarity" relation $\sim$ defined by the rule:

$$h\ [S \to I_X]\ h' \wedge a\ [T(S)]\ a' \Rightarrow \langle h, a \rangle \sim \langle h', a' \rangle$$

The equivalence class of $\langle h, a \rangle$ under $\sim^*$ is denoted by $\langle\!| h, a |\!\rangle$.

The functor action on morphisms is: $(\textsc{cls}\ F)(f\colon X' \to X)$ sends a family $\{t_g\}_{g:Y \to X}$ to the corresponding family $\{t_{(g';f)}\}_{g':Y \to X'}$.

The interpretation of **new** for such classes is:

$$\begin{aligned}[\![\mathbf{new}\ C\ P]\!]_X(u) &= i \cdot [\![P]\!]_X(u)[h](m) \cdot d \\ &\quad \text{where } \langle\!| h, i, m, d |\!\rangle = [\![C]\!]_X(u)[\mathrm{id}_X\colon X \to X]\end{aligned}$$

It makes use of the expansion morphism and the allocation and deallocation opertions rather directly.