On Algebraic Specifications of Object-Oriented Programs [*]

Sophia Drossopoulou
Imperial College London
S.Drossopoulou@imperial.ac.uk

James Noble [†]
Imperial College London
kjx@mcs.vuw.ac.nz

Abstract

Algebraic specifications give succinct specifications for abstract datatypes, consisting of equations in terms of the values of the datatypes and the operations performed on them.

One of the prevailing motives in the object-oriented paradigm is that an object's behaviour is characterized by the way the object responds to messages sent to it. Thus, one would expect that the natural way to specify object-oriented programs would be through a form of algebraic specification.

Nevertheless, the prevailing approach to specifications of object-oriented programs is through a kind of Hoare logic: objects' behaviour is defined via an abstract implementation. Classes define (abstract) variables to hold their state; method definitions act on that state; and pre- and post- conditions and class invariants describe permissible states.

In this paper we argue that algebraic specifications for object-oriented programs would be desirable, we discuss the problems one faces in applying algebraic specifications to object-oriented programming, and suggest some potential solutions.

1. Introduction

Algebraic specifications were suggested in the 70s and 80s to provide precise and succinct specification for algebraic data types [11, 29, 23]. Algebraic data types describe the set of values that may be created through the application of the operations mentioned in the data type's signature. The specification then consists of a set of equations which mentions the operations from that signature.
For example, the algebraic data type stack could be given as in the first four lines of figure 1, with the obvious meaning for the operations empty, push, pop and top.

In an object-oriented language, objects are described through class or interfaces, which list the constructors and the public methods available on such objects. The *interface* of a stack would be pretty similar to the algebraic datatypes; such a class is given in figure 1 where the constructor, Stack(), would return an empty stack, and the public methods push,pop and top correspond to the operations push,pop and top from the algebraic specification. Of course, the class Stack has some internal state, represented by private instance variables which are not part of its interface.

```
sig Stack   =   empty : Stack;
                push :  int x Stack − >  Stack;
                pop :  Stack − >  Stack;
                top :  Stack − >  int
− − − − − − − − − − − − − − − − − − − − − − − − − − − − − −

interface Stack {    public void push(int);
                     public void pop();
                     public int top();    }
```

Figure 1. Stack as an algebraic data type and as an interface in an oo language

Algebraic specifications consist of sequences of equations, which involve expressions of the data type being specified; thus they usually only involve operations from the particular abstract data type. For example, the algebraic specification of a stack would consist of the two equations at the top of figure 2. This specification is, in our view, the most elegant and succinct specification possible. Most importantly, it is in terms of the stack itself, and *does not refer to any entities external to the stack.*

In the philosophy of the object-oriented paradigm, encapsulation is paramount [18, 28, 24], and objects are meant to be understood in terms of the messages they exchange, and *not* in terms of their state (the values held in their instance variables). Dan Ingalls, for example, gives a *polymorphism* principle as "*A program should specify only the behavior of objects, not their representation*" [18]; Ungar and Smith likewise name this the *behaviorist* approach [28]. Thus, one would expect an OO specification of a stack to follow the flavour of the algebraic specification. Nevertheless, the style currently pursued in specifications of object-oriented programs does not live up to that philosophy. Instead, it makes use of model fields, which are an abstract representation of the object's state, that is, a *high level implementation* of the object's underlying behavioural specification [5, 19].

```
∀i : int, s : Stack :
                  pop(push(i, s))   =   s;
                  top(push(i, s))   =   i;
− − − − − − − − − − − − − − − − − − − − − − − − − − − − − −

          { ∅ }      s = newStack()     { s.is = [] }
     { s.is = ss }      s.push(i)       { s.is = i :: ss }
{ s.is = i :: ss }      s.pop()         { s.is = ss }
{ s.is = i :: ss }    j := s.top()      { j = i }
```

Figure 2. Specification of a stack in the algebraic style and the model based style

* Work partially supported by the EPSRC and RSNZ Marsden fund

† Permanent Address: School of Mathematics, Statistics, and Computer Science, Victoria University of Wellington, Box 600, Wellington, New Zealand.

For example, in a model-based system [3, 4, 26, 30] the specification of a stack would be in terms of a model field is, which would be a list of integers, with the concatenation operation ::, and the empty sequence indicated by []. Then, the specification would be similar to that given in figure 2. The Hoare triple, *e.g.,* $\{\ \mathsf{s.is} = \mathsf{i} :: \mathsf{ss}\ \}\ \ \mathsf{s.pop()}\ \ \{\ \mathsf{s.is} = \mathsf{ss}\ \}$ expresses that in a state where the model field is is a sequence that consists of i followed by ss, then, after execution of s.pop the model field would contain the sequence is.

In our opinion, this specification is *indirect*, in that it specifies the stack in terms of a further entity, the list, and in that it uses the model field, which is an abstract representation of the stack's state. Relationships between different operations such as push and pop are not explicit: rather they are mediated through this abstract implementation. As a result, the model-based specification does not express the intuition lying behind the stacks. What we would like to express is the following four properties, where **P3** is probably the most advanced, and difficult to express.

P1 Pushing an element into the stack, then applying some other operations which do not push nor pop onto the stack, and then applying top() on that stack will give me that particular element.

P2 Pushing an element onto the stack, then applying some other operations which do not push nor pop onto the stack, and then applying pop() on that stack will give me the *original* stack.

P3 Reading the top() of a stack does not have any side-effect.

P4 Pushing an element onto the stack, then applying some other operations which do not affect, and are not affected by the contents of the stack, and then applying top() on that stack has the same effect as just doing these operations.

Some attempts have been made to use algebraic specifications for OO languages, [15, 16, 14, 7] however these have concentrated on building tools, particularly for testing [27, 8, 17, 31]. To our knowledge, no work has yet tackled the fundamental questions of how best algebraic specifications themselves should be adapted for object-oriented systems; how object-oriented systems should be defined using such systems; and — most crucially — what such specifications would actually mean.

It is not without reason that OO specifications do not follow the algebraic approach. Namely, in adapting algebraic specifications to the imperative paradigm one has to tackle the following issues:

Equality In the algebraic datatypes setting we make use of powerful equality operators, *e.g.,* we use equality of stacks. This is possible because the meaning of an expression is the expression itself, and there is no global state that it depends on. However, in the imperative OO setting, the meaning of an expression is "context dependent", it depends on values stored in the stack and heap, which are not directly reflected in the expression itself.

Thus, it is unclear what is the meaning of equality in the imperative oo setting. For example, for two stacks s and s′, what does $\mathsf{s} = \mathsf{s}'$ mean? Does it mean that s and s′ point to the same stack, or that the two stacks contain the the same elements, or that they contain equal elements?

Intermediate Execution In the algebraic datatypes setting an expression describes an execution in its entirety. In the imperative OO setting, however, operations on an object do not immediately follow one another; we need to have some guarantees about an object's behaviour when *e.g.,* a push is followed by a sequence of messages on other objects, and then followed by a pop on the first object.

With this comes the need to characterize how operations interfere. For example, if we modify an object pushed onto the stack, does that change affect the stack? Alternatively, does pushing an object onto a stack change the object (if it can be seen from outside)?

Side-effects In the algebraic data types setting, operations just return values. But in the imperative, object oriented setting, methods can return values, *and* have side-effects. Both the return value and the side-effect need to be specified.

Point of view When we talk of equality, there is also the issue from whose point of view we are talking. For example, when we talk of "does not affect", there is the issue of how do we know it does not affect, and "how far does it not affect". In an OO program there are, implicitly, several viewpoints, and we shall be able to write better specifications if we can make these viewpoints explicit.

We have tackled these issues and have tried several approaches on cases studies, including, sets, stacks, dictionaries, and a registry of students with their average marks. None of the approaches, so far is completely satisfactory.

Nevertheless, we believe that the work is worth pursuing. We have identified major problems, and have identified some approaches to solving these. We believe that algebraic specifications are worthwhile, and that further research is necessary, and would benefit from our preliminary findings.

In the remainder of the paper we present our approach to the application of the algebraic specifications philosophy to the object-oriented paradigm: The equational approach, in section 2 is the most straightforward object-oriented extension of classical algebric specifications [11]. The possible worlds approach, in section 3 refies the notion of program state in terms of "worlds", and then allows specifications to describe hypothetical worlds and objects that are not be reached or part of the program being specified. Finally in section 4 we conclude and describe further work.

2. The equational approach

In this approach, we allow the specifier to define very powerful comparison operators, based on the equality of basic values (*i.e.,* integers, booleans, addresses), and universal quantification over messages. We also allow these messages to happen in different states of the execution.

The specification is written in terms of rules, where the premises of the rule describe some execution, and the conclusion contains equations in terms of these powerful equality operations.

In the following, we use exp as a metavariable for a sequence of expressions; we do not allow callbacks, *i.e.,* we require methods in the interface of a class do not call other methods from that interface. the meaning of "does not contain" is that the execution will not call push not pop on any alias of s. The operation $\equiv$ is the basic equality that compares basic values — in this case integers. Based on this basic equality we sill define more complex equivalence operations that compare two structures in (potentially different) runtime states.

2.1 Expressing P1

$$\text{(S}\textsc{tack}\text{E}\textsc{q}\text{1)}$$

$$\frac{\begin{array}{c}\mathsf{exp\ does\ not\ cause\ s.push(\_)\ \ nor\ \ s.pop()}\\ \mathsf{s.push(i);\ exp;\ j = s.top()}\end{array}}{\mathsf{i} \equiv \mathsf{j}}$$

The rule StackEq1 is satisfied, if for all possible runtime states χ, the execution of the sequence $\mathsf{s.push(i);\ exp;\ j = s.top()}$ in χ returns a value equal to that of j, provided that exp did not *cause* s.push() nor s.pop() to be executed.

$$\textsc{StackEq1}\ \text{satisfied, iff} \left\{ \begin{array}{l} \forall\chi, \forall \mathsf{exp}: \\ \mathsf{s.push(i);\ exp; s.top()}, \chi \rightsquigarrow \mathsf{exp;\ s.top()}, \chi_1 \ \rightsquigarrow^{\overline{\alpha}}\ \mathsf{s.top()}, \chi_2 \rightsquigarrow \mathsf{v}, \chi' \\ \{\overline{\alpha}\} \cap \{\mathsf{s.push(\_), s.pop()}\ \} = \emptyset \end{array} \right\} \implies \chi(\mathsf{i}) \equiv \mathsf{v}$$

$$\textsc{StackEq2}\ \text{satisfied, iff} \left\{ \begin{array}{l} \forall\chi, \forall \mathsf{exp}: \\ \mathsf{s.push(i);\ exp;\ s.pop()}, \chi \rightsquigarrow \mathsf{exp;\ s.pop()}, \chi_1 \ \rightsquigarrow^{\overline{\alpha}}\ \mathsf{s.pop()}, \chi_2 \rightsquigarrow \mathsf{v}, \chi' \\ \{\overline{\alpha}\} \cap \{\mathsf{s.push(\_), s.pop()}\ \} = \emptyset \end{array} \right\} \implies \mathsf{s}, \chi \equiv_{\mathsf{stack}} \mathsf{s}, \chi'$$

$$\textsc{StackEq3}\ \text{satisfied, iff} \left\{ \begin{array}{l} \forall\chi, \forall \mathsf{exp}: \\ \mathsf{exp}' = \mathsf{s.push(i);\ exp;\ s.pop()} \\ \mathsf{exp}', \chi \rightsquigarrow \mathsf{exp;\ s.pop()}, \chi_1 \ \rightsquigarrow^{\overline{\alpha}}\ \mathsf{s.pop()}, \chi_2 \rightsquigarrow \mathsf{v}, \chi' \\ \{\overline{\alpha}\} \cap \{\mathsf{s.push(\_), s.pop(), s.top()}\ \} = \emptyset \end{array} \right\} \implies \mathsf{exp}, \chi \equiv_{\mathsf{eff}} \mathsf{exp}', \chi$$

Figure 3. The precise meaning of StackEq1, and StackEq2

In more detail, execution of some expression exp, *does not cause* α iff it does not contain a sub-execution containing α. We assume that α is either a field access, or method call, or field assignment, *i.e.,* that $\alpha = \mathsf{id.f}$, or $\alpha = \mathsf{id.f} = \mathsf{id}'$, or $\alpha = \mathsf{id.m(id')}$ for identifiers (or addresses) id. We also assume runtime states χ, and a trace semantics[1], whereby $\exp, \chi \rightsquigarrow^{\overline{\alpha}} \mathsf{v}, \chi'$ means that in the runtime state χ the expression exp executes and produces value v and final states χ', and in the process, it reads/writes fields and calls methods according to the sequence $\overline{\alpha}$. We also write $\exp, \chi \rightsquigarrow \mathsf{v}, \chi'$, when we are not interested in the trace. Furthermore, we indicate erroneous executions by $\exp, \chi \rightsquigarrow \mathsf{error}, \chi'$, and we use the convention that the metavariable ranges over values, and is never error.

Then, assuming a deterministic operational semantics we say

$$\begin{array}{c} \mathsf{exp}, \chi \rightsquigarrow \mathsf{v}, \chi' \text{ does not cause } \overline{\alpha} \\ \text{iff} \\ \mathsf{exp}, \chi \rightsquigarrow^{\overline{\alpha'}} \mathsf{v}, \chi', \text{ and } \{\overline{\alpha}\} \cap \{\overline{\alpha'}\} = \emptyset. \end{array}$$

For identifiers id, we define their lookup in the runtime state as $\chi(\mathsf{id})$

We now can express the precise meaning of StackEq1 in figure 3.

2.2 Expressing P2

To express **P2**, we need a way to compare two stack pointers at different points of execution. We will say that

(StackEq2)

$$\frac{\begin{array}{c}\mathsf{exp\ does\ not\ cause\ s.push(\_)\ \ nor\ \ s.pop()} \\ \mathsf{s.push(i);\ exp; s.pop()}\end{array}}{\mathsf{s_{old}} \equiv_{\mathsf{stack}} \mathsf{s_{new}}}$$

In the above, as standard in Hoare logics, we use the subscripts old and new to indicate the stack as in the old state and in the new state.

The next question is the precise meaning of $\equiv_{\mathsf{stack}}$, which compares two stacks, possibly at *different* times of execution. We want to give an extensional definition of $\equiv_{\mathsf{stack}}$, where we compare the *behaviours* of the stacks, rather than their contexts.

In our first attempt, we define in figure 4 the relation $\equiv^1_{\mathsf{stack}}$, and require that a) any valid manipulations of s are also valid for s′, and b) after any number of manipulations of the two stacks in their respective states, the application of top() will return equal values. Note that for "equal values" we only require the comparison for basic values.

[1] For ease of notation, we require the operational semantics to be in terms of small steps, but also to be transitive.

We implicity close the relationship $\equiv^1_{\mathsf{stack}}$ with respect to symmetry, so that it is, as expected, an equivalence relation.

Obviously, determining the relation $\equiv^1_{\mathsf{stack}}$ is not necessarily straightforward. However, the primary aim of our work is to help write succinct and natural specifications, rather than provide tools for testing [15].

Another possible expression of equivalence of stacks, also given in figure 4, is to define $\equiv^2_{\mathsf{stack}}$ through the use of a smallest fixpoint construction, whereby we require that the application of top() gives equal results, and the application of push(...) or pop(..) again yield equivalent stacks in the sense of $\equiv^2_{\mathsf{stack}}$.

We believe that both $\equiv^1_{\mathsf{stack}}$ and $\equiv^2_{\mathsf{stack}}$ neatly capture the extensional equality of stacks, but while they express programmer's intuitions, their definition may be rather demanding. Furthermore, one might argue that these equality operations introduce a model through the back door. This is because both these definitions are effectively building up a model of the stack by canonicalising the stack via the pop and top operations. Each stack is reduced to a sequence starting with the top element, then those elements are compared.

We can easily refactor the definition into $\equiv^3_{\mathsf{stack}}$, also given in figure 4, to make this model-building clear. We canonicalise the stacks into sequences through the contents operation, and then compare the two sequences. This version of $\equiv^3_{\mathsf{stack}}$ has the advantage that it is easy to implement, but has the disadvantage that it is not extensional.

We can chose any of the above definitions to be the meaning of $\equiv_{\mathsf{stack}}$, and use it in figure 3 to give a precise meaning to StackEq2.

2.3 Expressing P3 and P4

In order to express the guarantee from **P4**, we will write

(StackEq3)

$$\frac{\mathsf{exp\ does\ not\ cause\ s.push(\_)\ \ nor\ \ s.pop()\ \ nor\ \ s.top()}}{\mathsf{s.push(\_);\ exp;\ s.pop()} \equiv_{\mathsf{eff}} \mathsf{exp}}$$

The meaning of $\equiv_{\mathsf{eff}}$ is, that execution of the two expressions has the same "effect" . This is defined in figure 4, where we use the term $\mathsf{z} \in \mathsf{dom}(\chi)$ to indicate the entities that can be seen in χ[2], and

[2] In a full formalization, these probably would be all entities that are named in the current scope, or which are reachable from the current scope through the public interfaces. Thus, we want to consider all students in the registry of a course (public interface), but we do not want to consider the nodes that implement the registry (private implementation).

$$s,\chi \equiv^1_{\mathsf{stack}} s',\chi' \text{ iff } \begin{cases} \forall \mathsf{exp},\ \text{with } s,s' \notin \mathsf{free(exp)}: \\ \mathsf{exp}[s/x],\chi \leadsto v,\_ \implies \mathsf{exp}[s'/x],\chi' \leadsto v',\_ \\ \left.\begin{array}{l} \mathsf{exp}[s/x],\chi \leadsto v_1,\chi_1 \\ s.\mathsf{top}(),\chi_1 \leadsto v,\_ \end{array}\right\} \implies \left\{\begin{array}{l} \mathsf{exp}[s'/x],\chi' \leadsto v_2,\chi_2 \\ s.\mathsf{top}(),\chi_2 \leadsto v',\_ \\ v \equiv v' \end{array}\right. \end{cases}$$

$$s,\chi \equiv^2_{\mathsf{stack}} s',\chi' \text{ iff } \begin{cases} s.\mathsf{top}(),\chi \leadsto v,\chi'' & \implies \quad s'.\mathsf{top}(),\chi' \leadsto v',\chi''',\ \ v \equiv v' \\ s.\mathsf{pop}(),\chi \leadsto v,\chi'' & \implies \quad s'.\mathsf{pop}(),\chi' \leadsto v',\chi''',\ \ s,\chi'' \equiv^2_{\mathsf{stack}} s',\chi''' \\ s.\mathsf{push}(z),\chi \leadsto v,\chi'' & \implies \quad s'.\mathsf{push}(z),\chi' \leadsto v',\chi''',\ \ s,\chi'' \equiv^2_{\mathsf{stack}} s',\chi''' \end{cases}$$

$$s,\chi \equiv^3_{\mathsf{stack}} s',\chi' \text{ iff } \mathsf{contents}_\chi(s) =_{\mathsf{seq}} \mathsf{contents}_{\chi'}(s')$$
where
$$\mathsf{contents}_\chi(s) \; = \; \begin{cases} v :: \mathsf{contents}_{\chi'}(s'), & \text{if } s.\mathsf{top}(),\chi \leadsto v,\chi', \text{ and } s.\mathsf{pop}(),\chi' \leadsto s',\chi'; \\ \epsilon, & \text{if } s.\mathsf{top}(),\chi \leadsto \mathsf{error},\chi'. \end{cases}$$
and $=_{\mathsf{seq}}$ is sequence equality.

$$\mathsf{exp},\chi \equiv_{\mathsf{eff}},\chi\mathsf{exp} \text{ iff } \begin{cases} \mathsf{exp},\chi \leadsto v_1,\_ \implies \exists v_2:\ \mathsf{exp}',\chi' \leadsto v_2,\_ \\ \left.\begin{array}{l} \mathsf{exp},\chi \leadsto v_1,\chi_1 \\ \mathsf{exp}',\chi \leadsto v_2,\chi_2 \end{array}\right\} \implies \left\{\begin{array}{l} dom(\chi_1) = dom(\chi_2) \\ \forall z \in \mathsf{dom}(\chi_1):\quad \mathcal{T}(z,\chi_1) = \mathcal{T}(z,\chi_2), \text{ and } z,\chi_1 \equiv_{\mathcal{T}(z,\chi_1)} z,\chi_2 \end{array}\right. \end{cases}$$

Figure 4. The meaning(s) of stack equivalence, and the meaning of equivalence of expressions

the notation $\mathcal{T}(z,\chi)$ to describe the type of z in the runtime state χ[3].

We say that exp,χ and exp',χ' are equivalent, if a) all valid executions of exp,χ are valid executions of exp',χ' [4], and b) if the outcomes of the execution of exp,χ and of exp',χ' are equivalent with respect to their types. We expect each type type to have an associated equivalence relation $\equiv_{\mathsf{type}}$. As for the definition of $\equiv_{\mathsf{stack}}$, we implicitly require $\equiv_{\mathsf{eff}}$ to be a symmetric relationship.

We now can express the precise meaning of STACKEQ3 in figure 3. We can express the guarantee from **P3**, in a similar way to **P4**, but omit it for here, for the sake of brevity.

2.4 Discussion

We believe that the above specifications neatly capture the extensional meaning of stack, even though the definitions of the equality operations may seem too complex. As we said earlier, these equalities might become simpler, if expressed in a more model-based style, *e.g.,* the relation $\equiv^3_{\mathsf{stack}}$ reflects a model. The temptation to work this way may explain some of the popularity of model based specifications, and the relative neglect of the algebraic approach.

An issue which needs to be tackled in any realistic setting is that in some sense, specifications have several different aspects. What if the pop and push operations maintained a counter of the number of applications of these operations, and in some aspects of the program the counters were relevant, and in others they were not? Which specifications should reflect that property and which should not?

Such aspects can neatly be tackled in the equational approach. Namely, we can define several different $\equiv$-relationships for stacks, some which make requirements on the values of the counters and some which do not.

3. The possible worlds approach

The difficulties with the equational approach have led us to investigate another approach, which we call *possible worlds*. As its name suggests, this approach makes more use of the worlds χ in the specification. We write rules which describe which worlds can possible be reached by a series of "moves" from the current worlds. Some of these worlds will be "possible worlds" because they will be part of the trace between the initial and final states: we hypothesize about the results of potential actions that the program does not actually take. Then, rather than making statements using strong equality operators, we make statements about the base-level operations in particular words. Instead of the oracle provide by the equality operations our rules will constrain only the observable behaviours of the objects in our specification. In this approach, the only equality operator we permit ourselves is the unadorned $\equiv$ that compares basic values.

The following rule, STACKTOPEQUALS, is a simple example to illustrate possible worlds and base-level equality.

(STACKTOPEQUALS)

$$\frac{\begin{array}{c} s'.\mathsf{equals}(s),\chi^{old} \leadsto \mathsf{true},\chi' \\ s'.\mathsf{top}(),\chi^{old} \leadsto v,\chi'' \end{array}}{s.\mathsf{top}(),\chi^{old} \leadsto v,\chi^{new}}$$

This rule states that if two stacks are equal in the old world χ^{old}, then the result of executing top on either of them will be the same. The "possible worlds" here are the worlds χ' and χ'' which the system would reach were it to carry out the hyppothetical top and equals operations involving s'. We read rules such as STACKEQUALS1 as: "If, in χ^{old}, you could have executed $s'.\mathsf{equals}(s)$ and the result would have been true, and if in χ^{old}, you could have executed

[3] In a full formalization, maybe the use of a static environment, and the use of static types for z would be better.

[4] In the definition given in figure 4, we know that the first execution is valid because it returns $v_1 \neq \mathsf{error}$, and because v_1 is a value, the execution cannot continue. Similar arguments apply to the second execution.

$\mathsf{s'.top()}$ and the result would have been v, then you can actually execute $\mathsf{s.top()}$ from χ^{old}, reaching χ^{new} and returning the result v".

3.1 Base-level Equality

The rule to define **P1** in possible worlds is the same as in the equational approach, because STACKEQ1 does not depend on a strong equality operator.

P2 is a different case because it relies crucially on such an operator to compare the old and new stacks across the top operation. To adess this, we can introduce a new *base-level* operation, equal: of course, most object-oriented programming languages require objects to provide an equals operation, so this is not an imposition in practice. The intuition behind the equals operation is that it should maintain the following property:

P5 Two stacks are equal if both are empty; once the same object has been pushed onto two equal stacks; or after two equal stacks have been popped.

which we can then provide algebraic rules to express. Asserting empty stacks are equal requires no possible worlds:

(STACKEQUALSEMPTY)

$$\frac{\begin{array}{c}\mathsf{exp}, \mathsf{exp'}\ \text{ do not cause }\ \mathsf{s.push(\_)}\text{ or }\mathsf{s.pop()}\\ \mathsf{exp}, \mathsf{exp'}\ \text{ do not cause }\ \mathsf{s'.push(\_)}\text{ or }\mathsf{s'.pop()}\end{array}}{\begin{array}{c}\mathsf{s = newStack(); exp; s' = newStack(); exp's.equals(s')}, \chi^{old}\\ \rightsquigarrow\ \ \mathsf{true}, \chi^{new}\end{array}}$$

Rules about commuting equals through push

(STACKEQUALSPUSH)

$$\frac{\begin{array}{c}\mathsf{s.equals(s')}, \chi^{old} \rightsquigarrow \mathsf{true}, \chi'\\ \mathsf{exp}, \mathsf{exp'}\text{ do not cause }\ \mathsf{s.push(\_)}\text{ or }\ \mathsf{s.pop()}\\ \mathsf{exp}, \mathsf{exp'}\text{ do not cause }\mathsf{s'.push(\_)}\text{ or }\ \mathsf{s'.pop()}\end{array}}{\mathsf{s.push(i); exp; s'.push(i); exp'; s.equals(s')}, \chi^{old} \rightsquigarrow \mathsf{true}, \chi^{new}}$$

and pop need a single possible world, to capture the notion that stacks are equal in the old world.

(STACKEQUALSPOP)

$$\frac{\begin{array}{c}\mathsf{s.equals(s')}, \chi^{old} \rightsquigarrow \mathsf{true}, \chi'\\ \mathsf{exp}, \mathsf{exp'}\text{ does not cause }\mathsf{s.push(\_)}\ \text{ or }\ \mathsf{s.pop()}\\ \mathsf{exp}, \mathsf{exp'}\text{ does not cause }\mathsf{s'.push(\_)}\ \text{ or }\ \mathsf{s'.pop()}\end{array}}{\mathsf{s.pop(); exp; s'.pop(); exp'; s.equals(s')}, \chi^{old} \rightsquigarrow \mathsf{true}, \chi^{new}}$$

Note that because the equational approach from section 2, relies on equational oracles, it does not need a base-level equals operation[5].

We now revisit the properties of the stack to see how they can be expressed *without* these metatheoretic equivalences in the possible worlds approach.

3.2 Expressing P2

To continue in this style requires us to not only use possible worlds but also to hypothesize about possible or potential *objects* in such worlds: this allows us to state rules to capture coniditions (such as **P2**, **P3**, and **P4**) which range across multiple different worlds — something we can also do directly in equational approach but not in a strict behavioural approach. We write $\mathsf{s} : \mathsf{Stack} \in \chi$ in specifications to hypothesize about the existence of an object o in world χ.[6] Then, we use trace exclusions to *project* these hypothetical objects forward into possible worlds.

Thus, the STACKEQUALS2 rule addresses **P2** by hypothesizing a second, ghost stack $\mathsf{s'}$ in χ^{old} and requiring a) that it is equal to our stack s; and b) that it is not changed between χ^{old} and χ^{new}, then asserting that the two stacks are still equal after $\mathsf{s.top()}$.

(STACKEQUALS2)

$$\frac{\begin{array}{c}\mathsf{s'} : \mathsf{Stack} \in \chi^{old}\\ \mathsf{s.equals(s')}, \chi^{old} \rightsquigarrow \mathsf{true}, \chi'\\ \mathsf{s.equals(s')}, \chi^{new} \rightsquigarrow \mathsf{true}, \chi''\\ \chi^{old} \rightsquigarrow \chi^{new}\text{does not cause }\mathsf{s'.push(\_)}\ \text{ or }\ \mathsf{s'.pop()}\\ \mathsf{exp}\text{ does not cause }\mathsf{s.push(\_)}\ \text{ or }\ \mathsf{s.pop()}\end{array}}{\mathsf{s.push(i);\ exp; s.pop()}, \chi^{old} \rightsquigarrow \mathsf{void}, \chi^{new}}$$

In the above, the assertion "$\chi^{old} \rightsquigarrow \chi^{new}$ does not cause $\mathsf{s'.push(\_)}$ or $\mathsf{s'.pop()}$" means that there exists a transition that takes the system from world χ to world χ', and does not cause $\mathsf{s'.push(\_)}$, or $\mathsf{s'.pop()}$. In general, we define $\chi^{old} \rightsquigarrow \chi^{new}$does not cause $\overline{\alpha}$ iff $\exists \mathsf{exp}, \mathsf{v} :\ \mathsf{exp}, \chi^{old} \rightsquigarrow_{\overline{\alpha'}} \mathsf{v}, \chi^{new}$, and $\{\ \overline{\alpha'}\ \} \cap \{\overline{\alpha}\} = \emptyset$.

3.3 Expressing P3

We can express P3 in a similar manner:

(STACKEQUALS3)

$$\frac{\begin{array}{c}\mathsf{s'} : \mathsf{Stack} \in \chi^{old}\\ \mathsf{s.equals(s')}, \chi^{old} \rightsquigarrow \mathsf{true}, \chi'\\ \mathsf{s.equals(s')}, \chi^{new} \rightsquigarrow \mathsf{true}, \chi''\\ \chi^{old} \rightsquigarrow^{*}\ \chi^{new}\text{does not cause }\mathsf{s'.push(\_)}\ \text{ or }\ \mathsf{s'.pop()}\end{array}}{\mathsf{s.top()}, \chi^{old} \rightsquigarrow \mathsf{v}, \chi^{new}}$$

again stating if two stacks are equal, calling top should not change that.

3.4 Expressing P4

The final condition is again a framing condition which does not rely on projection, and only peripherally on possible worlds. Rather than relying on another oracle ($\mathsf{exp'} \equiv_{\mathsf{eff}} \mathsf{exp}$) we can say talk about the transitions between (real) worlds by naming worlds explicitly.

(STACKEQUALS4)

$$\frac{\begin{array}{c}\mathsf{s.push(i); exp; s.pop()}, \chi^{old} \rightsquigarrow \mathsf{v}, \chi^{new}\\ \mathsf{exp}\text{ does not cause }\mathsf{s.push(\_)}\text{ or }\ \mathsf{s.pop()}\text{ or }\mathsf{s.top()}\end{array}}{\mathsf{exp}, \chi^{old} \rightsquigarrow \mathsf{v'}, \chi^{new}}$$

This rule says is that if you begin in χ^{old}, push something onto s, do some non-interfering operations then pop s and arrive at χ^{new}, you could equally have got from χ^{old} to χ^{new} without any stack operations — effectively allowing us to eliminate the stack operations.

We could use this technique to supplement the projection-based rule for P3 with a similar elimination rule for top:

[5] Such an equation can be provided, and its specification is trivial:

(STACKEQ5)

$$\frac{\mathsf{s} \equiv_{\mathsf{stack}} \mathsf{s'}}{\mathsf{s.equals(s')} \equiv \mathsf{true}}$$

thus demonstrating how the oracular $\equiv_{\mathsf{stack}}$ operation is really doing all the work in the equational approach!

[6] The assertion $\mathsf{s} : \mathsf{Stack} \in \chi$ differs from the assertion $\mathsf{s} \in \mathsf{dom}(\chi)$ from section 2, in that the former talks about *all* objects in the runtime state χ, while the latter only talks of the *public* objects in χ. In terms of our earlier examples, the former also contains the node objects of a registry, while the latter does not.

(StackEquals3')

$$\frac{\mathsf{exp; s.top()}, \chi^{old} \leadsto \mathsf{v}, \chi^{new}}{\mathsf{exp}, \chi^{old} \leadsto \mathsf{v}', \chi^{new}}$$

4. Discussion & Conclusion

The possible worlds approach, and its treatment of an object-oriented program as transitions between worlds that both change state and calculate values, allows objects to be given specifications that do not rely either on abstract models or complex equivalences. Unfortunately, that is only the start of the story, not the end! There are many important issues our basic treatment cannot resolve.

4.1 Inheritance

Object-oriented programming languages generally provide an inheritance mechanism: OO specification languages follow this with support for "specification inheritance" [9, 6, 3, 4, 22]. Rules for specification inheritance are often complex, generally seeming to duplicate rules of inheritance in programming languages: maintaining the integrity of pre- and post- conditions and invariants over the state in their model fields. This complexity strengthens the argument that such models are abstract implementations, rather than specifications [19]. In the algebraic approach, we do not expect such inheritance anomalies: specifications defined solely in terms of the external behaviour of objects can be inherited and combined straighforwardly.

4.2 Reference vs Value Semantics

A more subtle issue is the distinction between reference and value semantics, particularly for equality [2], but also for other operations. In writing either equational definitions, or the specifications of base-level equality operations, programmers need to decide on (and distinguish between) reference, value, or more exotic modes for communication between objects.

So far, our specifications have been mostly in terms of reference semantics. Rule StackEq1 for example compares the object pushed on and popped of the stack simply with reference equality $\equiv$. We can, however, use hypothetical objects and possible worlds to write a specification that demands value semantics: effectively that objects are copied in and out of the stack — copying up to a particular equals operator.

(StackValue1)

$$\frac{\begin{array}{c}\mathsf{o}' : \mathsf{Object} \in \chi^{old} \\ \mathsf{o}'.\mathsf{equals(o)}, \chi^{old} \leadsto \mathsf{true}, \chi' \\ \mathsf{o}'.\mathsf{equals(o'')}, \chi^{new} \leadsto \mathsf{true}, \chi'' \\ \forall \mathsf{exp}' : \mathsf{exp}', \chi^{old} \leadsto^* \mathsf{v}, \chi^{new} \\ \mathsf{exp}' \text{ does not involve } \mathsf{o} \\ \mathsf{exp} \text{ does not cause } \mathsf{s.push(\_)} \text{ nor } \mathsf{s.pop()}\end{array}}{\mathsf{s.push(o);\ exp; s.pop()}, \chi^{old} \leadsto \mathsf{o}'', \chi^{new}}$$

Here the hypothetical object o' acts as a proxy for the state of o in χ^{old}; the "does not involve" clause aims to project this object forward to χ^{new}, where it can be compared with whatever object (o'') is popped off the stack.

4.3 External Iterators and Separate Interfaces

External iterators, and other objects which provide secondary interfaces to objects, can be challenging to many schemes which aim to treat objects as encapsulated behind their interfaces [20, 21, 25]. There are typically two main problems here: first the need to grant controlled access to private state in a specification or programming language, and second, the requirement to identify that objects accessing such shared state must be treated as part of the primarly abstraction, so that the messages they receive may inspect or alter the state of the main object. Because behavioural, algebraic specifications do not explicitly declare shared state, they are not susecptible to the first problem, but the second still applies.

To take a simple example that illustrates both the problem and our solution imagine enhancing the stack with a getIter() operation that returns an iterator; the iterator supports a item() operation that just returns the top() of the stack. We can specify this as follows:

(StackSimpleIter)

$$\frac{\begin{array}{c}\mathsf{s.getIter()}, \chi'' \leadsto \mathsf{i}, \chi' \\ \mathsf{exp}, \chi' \leadsto^* \mathsf{v}, \chi^{old} \\ \mathsf{s.top()}, \chi^{old} \leadsto \mathsf{o}, \chi'''\end{array}}{\mathsf{i.item()}, \chi^{old} \leadsto \mathsf{o}, \chi^{new}}$$

the key to this rule being the "backwards-looking" specification that finds the iterator creation operation (in a prior world χ''), and uses that to determine the stack s to which the iterator belongs.

4.4 Classes and Shared State

A similar problem can exist if we wish to model state that crosscuts objects, such as class (static) variables. We can model classes and their interfaces by reifying them as standalone prototypical objects [1], but class methods called on instances will by nature look odd: the specifications must disregard the instance upon which static messages are called. The following specification states that the result of calling colour() on a widget is the argument passed in to the most recent call to setColour() on some other widget — no matter which widgets are involved.

(WidgetClassColour)

$$\frac{\mathsf{exp} \text{ does not cause } \mathsf{w}'.\mathsf{setColour(c')}}{\mathsf{w.setColour(c); exp; w''.getColour()}, \chi^{old} \leadsto \mathsf{c}, \chi^{new}}$$

4.5 Contracts

It is also interesting to consider how an algebraic approach could be extended to support interobject contracts [13, 12]. The iterator and class-wide state examples demonstate that specifications can be written with surprisingly little respect for object encapsulation — that is, that these specifications can cut across messages sent to multiple objects. In the case of contracts, however, we need to require that groups of objects interact in particular ways whenever certain events occour.

The canonical example is the observer pattern: whenever notify() is called on a subject, the subject must itself call update() on all of its observers [10]. The specifications shown in this paper do not permit us to write such a contract, generally because the calls to update() must occur *during* the excution of notify() — so notify() should not return until all updates are complete. We expect a relatively straightforward extension, allowing us to consider method send and return events individually, will support this kind of contract specifications.

4.6 Worlds and Objects

The final, philosophical question is to return again to the question of the worlds in our specifications. In section 2.1 we suggested that χ was the *runtime state* of the system, and was affected by field reads and writes and method calls. In retrospect, that seems like an odd defintion, as none of our specifications have discussed reading or writing fields, and none of the specifications involving methods have directly described any change of this state: they simply refer to arguments or return values of previous message calls. This suggests an alternative description of the state:

The state of the world is the history of all messages sent to all objects since the start of the system.

That is, the world (or its state, or the state of a system implementing such a specification) keeps track of all the messages since the system is started (the "big bang"). It is very tempting, in an object-oriented system, to hypothesize the following:

The state of **an object** is the history of all messages sent **to that object** since the start of the system.

Unfortunately, while we permit specifications for iterators, external interfaces, class-wide effects and so on, this hypothesis does not hold: since we can write a specification such that any message sent anywhere can affect the result of any other message The best we can do, it seems, is to return to this:

The state of an object is the history of all messages sent **to all objects** since the start of the system.

Algebraic specification languages such as Spec♯ and JML are adopting ownership systems [3, 4] to deal with this issue. Given that our specifications do not have any implementations, and we have no representation objects we need to protect, it appears the algebraic approach does not need an explicit ownership system to protect invariants induced by the specifications (after all, we do not write state-based invariants). Rather, we could consider restrictions on the kind of behaviour we choose to specify — not to preserve correctness, but rather to ensure that our so-called "object-oriented" specifications are in practice object-oriented.

4.7 To Conclude

Are algebraic specifications for object-oriented programs *desirable*? Yes, they are; they reflect best the intentions and the philosophy of the paradigm.

Are algebraic specifications for object-oriented programs *possible*? We believe they are, and practical experience with relatively informal algebraic specifications, along with our small examples, show they can be written and have promise [15, 16, 27, 8, 17].

But much remains to be done. Our further work includes the consideration of more case studies, the development of syntactic sugar to make idioms easier to read and write, and the full development of the semantics and conceptual foundations for an algebraic model of specifications for object-oriented systems.

References

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

[2] H. G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4), Oct. 1993.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6), 2004.

[4] P. Chalin, J. Kiniry, G. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO Proceedings*, 2005.

[5] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract: Research articles. *S—P&E*, 35(6), 2005.

[6] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE Proceedings*, 1996.

[7] B. Dölle and W. Dosch. Transforming functional signatures of algebraic specifications into object-oriented class signatures. In *APSEC Proceedings*, 2005.

[8] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM ToSEM*, 3(2), 1994.

[9] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *FSE Proceedings*, 2001.

[10] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[11] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1), 1978.

[12] J. He, Z. Liu, and X. Li. Contract-oriented component software development. Technical Report 276, UNU/IIST Macau, 2003.

[13] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *OOPSLA Proceedings*, 1990.

[14] J. Henkel. *Discovering and Debugging Algebraic Specifications for Java Classes*. PhD thesis, University of Colorado at Boulder, 2004.

[15] J. Henkel and A. Diwan. Discovering algebraic specifications for java classes. In *ECOOP Proceedings*, 2003.

[16] J. Henkel and A. Diwan. A tool for writing and debuggin algebraic specifications. In *ICSE Proceedings*, 2004.

[17] M. Hughes and P. D. Stotts. Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In *ISSTA Proceedings*, 1996.

[18] D. H. Ingalls. Design principles behind Smalltalk. *BYTE Magazine*, Aug. 1981.

[19] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[20] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte.a. Iterators revisited: Proof rules and implementation. In *FTfJP Proceedings*, 2005.

[21] D. A. Lamb. Specification of iterators. *IEEE TSE*, 16(12), 1990.

[22] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report TR-06-20, Iowa State University, 2006.

[23] J. McLean. A formal method for the abstract specification of software. *JACM*, 31(3), 1984.

[24] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[25] J. Noble. Iterators and encapsulation. In *TOOLS Proceedings*, 2000.

[26] M. Parkinson. When separation logic met Java. in 2006. In *FTfJP Proceedings*, 2006.

[27] P. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *XP/Agile Universe*, 2002.

[28] D. Ungar and R. B. Smith. SELF: the Power of Simplicity. In *OOPSLA Proceedings*, 1987.

[29] Y. Wang and D. L. Parnas. Simulating the behaviour of software modules by trace rewriting. In *ICSE*, 1993.

[30] A. C. Wills. *Formal Methods applied to Object-Oriented Programming*. PhD thesis, University of Manchester, 1992.

[31] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE Proceedings*, 2004.