First Steps Towards Fully Abstract Semantics for Object-Oriented Languages[1]

P. M. YELLAND

University of Cambridge Computer Laboratory,
New Museums Site,
Pembroke Street,
Cambridge CB2 3QG,
U.K.

E-Mail: pmcy@cl.cam.ac.uk

Abstract

A number of denotational models have been proposed for object-oriented languages. Authors of more recent models have expressed dissatisfaction with the lack of "abstractness" in earlier ones. They claim that these earlier models describe details of objects which are invisible to an external observer—in short, that they are not *fully abstract*. In this paper, we present a formal characterization of the visible behaviour of objects. We show that using a natural full abstractness criterion based on this definition, even more recent models of object-oriented languages are unnecessarily "concrete." We go on to present a semantics for a very simple object-oriented language based on projections of state-transition graphs for programs, and demonstrate that it is fully abstract.

1 INTRODUCTION

The use of object-oriented languages like Smalltalk-80, C++, CLOS and so on, has grown markedly in recent years, and promises to become even more widespread in future [Ren82]. The theoretical aspects of these languages have received rather less attention. However, there have been three notable attempts to devise formal denotational models for object-oriented languages.[2]

The earliest of these models, by Mario Wolczko [Wol87], draws heavily on the operational definition of Smalltalk-80 given by the description of the "Smalltalk Virtual Machine" in [GR83]. His model resembles the Virtual Machine in many respects—for example, it replicates the message lookup mechanism used by the machine to carry out message-passing. This makes it easy to verify its correctness, but also means that it embodies more details of the machine implementation than are strictly necessary to explain the semantics of the language. In fact, Wolczko

[1]This work supported by the S.E.R.C.

[2]The languages dealt with in this paper are all *sequential*. Concurrent object-oriented languages and their semantics, as described in [ADKR86] for example, are beyond the scope of our discussion.

makes only mild claims for the abstractness of his semantics, merely saying, for example, that it "hides the garbage-collection strategy."

In [Kam88], Samuel Kamin takes a rather stronger line on the issue of abstractness. He shows in particular that it is possible to model message-passing in Smalltalk without replicating the lookup mechanism.

To our knowledge, the most recent semantics for object-oriented languages is given by Uday Reddy in [Red88]. Reddy points out that Kamin's model is still unnecessarily detailed. He demonstrates that the semantics he proposes is more abstract than Kamin's (and Wolczko's, by implication).

Both Reddy and Kamin take the view that the denotations of objects should only be as detailed as is necessary to explain their externally observable behaviour—a property commonly known as *full abstraction* [Plo77] [Sto88] [BCL85] [Mey88].[3] The semantics of Wolczko, Kamin and Reddy represent progress towards the fulfillment of this condition. However, none of them satisfy it entirely.

To show why this is, we will concentrate on Reddy's semantics, which—as he himself shows—is more abstract than Kamin or Wolczko's. In the next section, we will present a simple object-oriented language. Using the same methods used by Reddy in his paper, we will produce a denotational model for this language. Next, we will develop a formal condition which characterizes full abstractness in a semantics, and show how Reddy's semantics fails to satisfy this criterion. We will then produce a fully abstract semantics based upon equivalence classes of terms, formed using an "observational congruence" derived from Reddy's semantics. Finally, using projections of state transition graphs for programs, we will construct a model isomorphic to it, without recourse to such a congruence.

The example object-oriented language used in our discussion is very primitive. It lacks many features commonly associated with "fully fledged" object-oriented languages, such as recursion, parameter-passing, dynamic object creation, partial operations and inheritance. However, it does embody three important aspects of the object-based approach to programming:

- Entities in a system are represented by objects with state, referred to exclusively by name. This latter feature means that objects may be *shared* by other objects.
- Some of the state of an object may be concealed, so that only partial information concerning it is available to other objects.
- There are operations which change the state of objects. A direct consequence of the previous point is that the effects of such an operation may only be partly visible.

[3] The question of terminology in this area of programming language semantics is a little vexed. In this paper, we will use nomenclature from [Sto88].

Support for the view that these features are central to object-oriented programming may be found in [GM87]. Since we can show that existing models fail to give abstract representations to objects in this "minimalist" object-oriented language, we can therefore claim that they will fail to be abstract for almost *all* languages which purport to be object-oriented. Correspondingly, by showing how abstract models may be produced for this language, we hope to lay the foundation for abstract descriptions of more complex object-oriented programming languages.

2 A SIMPLE OBJECT-ORIENTED LANGUAGE

In the language introduced in this section, programs consist of a finite sequence of object-definitions. Each object is defined individually—not indirectly, using classes, as is the case with most object-oriented languages. Objects defined in a program are indexed with successive natural numbers, starting with 1 at the beginning of the program. Each object has a finite set of variables, which may contain one of the boolean values true or false. They are initialized explicitly in the definition of the object.

An object also has two finite sets of operations. Operations of the first kind extract the value of a variable from an object. These operations are named after the variable whose value they return, and they are listed as "extractors" in definitions. It is important to note that not all the variables of an object need have a corresponding extractor. The second kind of operation is used to cause the object to change the contents of its variables. These operations are listed as "procedures," and their effects are defined by a finite sequence of statements. A statement may either:

- Transfer the contents of one variable to another.
- Apply an extractor to an object with a given index, and store the result in a variable.
- Apply another procedure to an object whose index is given.

We use the notation '$i.op$' in a statement to apply a procedure or an extractor named *op* to the object refered to by index i. *Such indices, occuring in the definition of an object, must be strictly less than the object's own index.* This restriction effectively bans cyclic chains of reference among objects.

Below is a formal syntax for the language. If i is an integer, then $\overline{i}$ will be its numeral representation. In the same way, $\overline{b}$ stands for the written representation of the boolean value b.

$\overline{i} \in \overline{\mathbf{N}}$ —numerals
$\overline{b} \in \overline{\mathbf{B}}$ —booleans
$v, e \in Id_v$ —variable identifiers
$p \in Id_p$ —procedure identifiers

program ::= *object-definition*; ...; *object-definition*
object-definition ::= **object** $\vec{i}$ **has** *vars-list exs-list procs-list*
vars-list ::= **variables** $v = \vec{b}, \ldots, v = \vec{b}$
exs-list ::= **extractors** $e, \ldots, e$
procs-list ::= **procedures** p **is** *stmt* ... p **is** *stmt*
stmt ::= $v \leftarrow v \mid v \leftarrow \vec{i}.e \mid \vec{i}.p \mid$ *stmt*; *stmt*

As an example of the use of this notation, the following program defines an object which behaves like a switch. The switch may either be on (*on* = **true**), or off (*on* = **false**). The state of the switch may be extracted, and a procedure *flip* is provided to invert it. The switch starts in the off position.[4]

object 1 has
variables *on* = **false**
extractors *on*
procedures *flip* **is** *on* $\leftarrow$ *not*(*on*)

3 A SEMANTICS IN THE STYLE OF REDDY

The language defined in the previous section bears a great similarity to Reddy's "ObjectTalk," as defined in [Red88]. In this section, we will use the techniques Reddy uses for ObjectTalk to produce a denotational semantics for our language. Since our language is much simpler than ObjectTalk, our treatment is correspondingly less complex.

Because our language is incapable of expressing infinite or non-terminating computations, and because all of the domain definitions we use may be solved in classical set-theory without problems of cardinality, we will use that as our semantical framework, rather than using, say, Scott domains.

We will use maps to represent many entities in the semantics. If η is such a map, then we use the usual notation $\eta[i_1 \leftarrow d_1, \ldots, i_n \leftarrow d_n]$ to produce a new map:

$$(\eta[i_1 \leftarrow d_1, \ldots, i_n \leftarrow d_n])i = \begin{cases} d_1 \text{ if } i = i_1, \\ \ldots, \\ d_n \text{ if } i = i_n, \\ \eta i \text{ otherwise} \end{cases}$$

The notation [] will represent an "empty" map—that is one which returns some "error" value when applied to any member of its domain. Its type will always be determined by the context. We write $[i_1 \leftarrow d_1, \ldots, i_n \leftarrow d_n]$ instead of $[][i_1 \leftarrow d_1, \ldots, i_n \leftarrow d_n]$. We use the notation $< a_1, \ldots, a_n >$ to denote the n-tuple with elements $a_1, \ldots, a_n$; π_i will name the i-th projection on such a tuple. We will

[4] In this example, we assume the existence of a primitive operation *not* on boolean values. We do not include its formal definition—it is quite simple, but would clutter the exposition.

omit injections and projections to and from disjoint sums. As usual, function-space construction associates to the right, function application to the left.

A detailed explanation of the sematics may be found in [Red88].

Semantic Domains

$i, j, n \in \mathbf{N}$ —the natural numbers
$b \in \mathbf{B}$ —the boolean values $\{\mathrm{tt}, \mathrm{ff}\}$
error —the error value

$\eta \in Env = \mathbf{N} \to (Obj + \{\mathbf{error}\})$
$\sigma \in State = (\mathbf{N} \times Id_v) \to (\mathbf{B} + \{\mathbf{error}\})$
$EMap = Id_v \to ((State \to \mathbf{B}) + \{\mathbf{error}\})$
$PMap = Id_p \to ((State \to State) + \{\mathbf{error}\})$
$Obj = EMap \times PMap$

Semantic Functions

P : *program* $\to (Env \times State)$
O : *object-definition* $\to (Env \times State) \to (Env \times State)$
V : *vars-list* $\to \mathbf{N} \to State \to State$
B : *exs-list* $\to \mathbf{N} \to EMap$
Q : *procs-list* $\to \mathbf{N} \to Env \to PMap$
S : *stmt* $\to \mathbf{N} \to Env \to State \to State$

Semantic Equations

$$P[\![\mathit{object\text{-}definition}_1; \ldots; \mathit{object\text{-}definition}_n]\!] = O[\![\mathit{object\text{-}definition}_n]\!](\ldots O[\![\mathit{object\text{-}definition}_1]\!] < [], [] > \ldots)$$

$$O[\![\textbf{object } \vec{i} \textbf{ has } \mathit{vars\text{-}list}\ \mathit{exs\text{-}list}\ \mathit{procs\text{-}list}]\!] < \eta, \sigma > = < \eta[i \leftarrow < B[\![\mathit{exs\text{-}list}]\!] i, Q[\![\mathit{procs\text{-}list}]\!] i\eta >], V[\![\mathit{vars\text{-}list}]\!] i\sigma >$$

$$V[\![\textbf{variables } v_1 = \vec{b}_1, \ldots, v_j = \vec{b}_j]\!] i\sigma = \sigma[< i, v_1 > \leftarrow b_1, \ldots, < i, v_j > \leftarrow b_j]$$

$$B[\![\textbf{extractors } e_1, \ldots, e_k]\!] i = [e_1 \leftarrow \lambda\sigma.\ \sigma < i, e_1 >, \ldots, e_k \leftarrow \lambda\sigma.\ \sigma < i, e_k >]$$

$$Q[\![\textbf{procedures } p_1 \textbf{ is } \mathit{stmt}_1 \ldots p_l \textbf{ is } \mathit{stmt}_l]\!] i\eta = [p_1 \leftarrow \lambda\sigma.\ S[\![\mathit{stmt}_1]\!] i\eta\sigma, \ldots, p_k \leftarrow \lambda\sigma.\ S[\![\mathit{stmt}_l]\!] i\eta\sigma]$$

$$S[\![\mathit{stmt}_1; \mathit{stmt}_2]\!] i\eta\sigma = S[\![\mathit{stmt}_2]\!] i\eta(S[\![\mathit{stmt}_1]\!] i\eta\sigma)$$
$$S[\![v \leftarrow v']\!] i\eta\sigma = \sigma[< i, v > \leftarrow \sigma < i, v' >]$$
$$S[\![v \leftarrow \vec{j}.e]\!] i\eta\sigma = \sigma[< i, v > \leftarrow \pi_1(\eta j) e\sigma]$$
$$S[\![\vec{j}.p]\!] i\eta\sigma = \pi_2(\eta j) p\sigma$$

4 ON EXTERNALLY OBSERVABLE BEHAVIOUR

4.1 The Nature of Observation

In the introduction, we talked of Reddy's semantics recording details which were not germane to the "observable behaviour" of objects. To put our objections in more precise terms, we must come to some rigorous definition of what we mean by "observable behaviour" in the context of object-oriented languages.

We begin by examining the concept of observation. It is helpful to bear in mind three questions:

1. Who is making the observation?
2. What is being observed?
3. How is the observation carried out?

With more conventional languages, the answers to these questions are fairly obvious. One observes the behaviour of a *program*, and to do this, one provides it with input, runs it, and then takes a note of the output produced. The observer is someone sitting at a computer terminal or similar peripheral. Most conventional languages have fairly recognizable input/output statements, so it is easy to see where interaction with the observer is taking place in a program.

The situation with object-oriented languages is not nearly so clear-cut. As Wolczko points out in his thesis [Wol88], input/output is not easily detected in most object-oriented languages. Indeed, in the case of Smalltalk, it is impossible to tell in general from the text of a program if it elicits any output at all.[5] In addition, a significant number of the class-definitions issued with every Smalltalk-80™ system never call for any I/O. So a notion of observation based upon I/O would be inappropriate for object-oriented languages.

To find a more useful definition of observation, we must look at the concern almost all object-oriented languages have with the re-use of software. Most proponents of object-oriented languages lay great store by the fact that they support the construction of self-contained, re-useable units of software [Cox83]. The standard class-definitions of Smalltalk-80 mentioned in the previous paragraph are intended to be incorporated by users of the system into their programs. In a very natural sense, the observer of a piece of software in an object-oriented language is a person who is writing another piece of software which makes use of it. Returning to the language we defined in section 2, we can say that observations are carried out by someone writing an object-definition. This answers the first of our questions.[6]

[5] I/O results as a side-effect of certain (dynamically-bound) messages sent to objects in the system.

[6] Readers familiar with the algebraic approach to concurrency will note the similarity between our notions of observation and those used in the "testing equivalences" of Hennessy and DeNicola [Hen88].

Turning to question 2, we note that when writing the definition of a new object, we are can elicit behaviour from any object which is accessible from it. Recall that all references from one object to another stem from the use of object indices in procedure statements, and that indices in procedures of object $n+1$ are drawn from the natural numbers 1 to n. So the writer is really observing a collection of objects—those with indices $1 \ldots n$. Henceforth, the term *system* will be used for such a collection of objects; a system is a finite collection of objects indexed by consecutive natural numbers, which includes an object with index 1. You will note that the definition of a system coincides with that we gave of a program, and we will use the semantic function P to compute the denotations of systems. The answer to question 2, then, is that observations are carried out on systems.

To see how observations are made, we must consider what the writer of a new object-definition can "see" directly, and how she or he might interact with the objects in a system. In our language, the writer cannot access the variables of other objects, but does have direct access to the variables in the object being defined. Their contents may be loaded explicitly by initialization, moved about by assignments, and used to record result of applying observers to other objects, whose states may themselves have been changed by the application of procedures. All these actions are carried out by executing statements. We can therefore use the following definition of observation to answer question 3:

> To observe a system from a new object, load all of the variables of that object with a set of values, execute some sequence of statements, and then take a note of the resulting contents of the variables.

We can summarize all the information needed for an observation as a triple $<< v_1, \ldots, v_m >, < l_1, \ldots, l_m >, < s_1, \ldots, s_n >>$, where $< v_1, \ldots, v_m >$ are the names of the new object's variables, $< l_1, \ldots, l_m >$ are the values to be loaded into them initially, and $< s_1, \ldots, s_n >$ are the statements to be executed. Call such an observation *legal* if the execution of its statements does not lead to an error.

Using the semantics of the previous section, we can now define the notion of observation formally:

Definition 1 *If S is a system containing definitions indexed $1 \ldots n$, then the result of applying a legal observation $obs = << v_1, \ldots, v_m >, < l_1, \ldots, l_m >, < s_1, \ldots, s_n >>$, written $obs(S)$, is a sequence of values $< l'_1, \ldots, l'_m >$ such that:*

$$l'_i = (S[\![s_1, \ldots, s_n]\!](n+1) < \eta, \sigma[< n+1, v_1 > \leftarrow l_1, \ldots, < n+1, v_m > \leftarrow l_m] >) < n+1, v_i >,$$

where $1 \leq i \leq m$, and $< \eta, \sigma >$ is the denotation of the system S.

4.2 Observational Equivalence and Abstract Semantics

Now that we have fixed upon a definition for observation, we can say when two systems have the same observable behaviour. This occurs when we cannot tell them apart using observations:

Definition 2 *Two systems S and S' are* observationally equivalent, $S \equiv_{obs} S'$, *iff for any observation obs legal on S, obs is legal on S' too, and $obs(S) = obs(S')$.*

At last, we can formalize the condition for a semantics to be fully abstract:

Definition 3 *A semantics with a semantic function P' on programs is* fully abstract *iff for any two systems S_1, S_2:*

$$S_1 \equiv_{obs} S_2 \text{ iff } P'[\![S_1]\!] = P'[\![S_2]\!]$$

Bear in mind that the notion of observational equivalence used on the left of the bi-implication uses the semantic functions of *Reddy's* semantics. If in their place we were to use the functions from the semantics defining P', then we would admit incorrect semantics,[7] such as one in which $S'[\![stmt]\!] = \lambda i, \eta, \sigma, < i, v > .\text{tt}$ and $P'[\![program]\!] =< \lambda i. < \lambda e\sigma.\text{tt}, \lambda p\sigma < i, v > .\text{tt} >, \lambda < i, v > .\text{tt} >$. By basing our definition on Reddy's semantics, we can ensure that the abstract semantics is correct. Note that we assume that Reddy's model is correct, even if it is not fully abstract.

4.3 Abstractness in the First Semantics

We can now show why the semantics in the style of Reddy is not fully abstract. Take the following two simple systems defining switches:

object 1 has
variables *on* = **false**
extractors *on*
procedures *flip* **is** *on* ← *not*(*on*)

and

object 1 has
variables *on* = **false**, *extraneous* = **false**
extractors *on*
procedures *flip* **is** *on* ← *not*(*on*); *extraneous* ← **true**

[7] That is, one which contradicts our intuitive operational understanding of the language.

It should be intuitively clear—and it is easy to show formally—that these two systems are observationally equivalent. In both cases, only the variable *on* is visible, and the only way to change the state of object 1 in both systems is by applying *flip*. The value of *on* is identical in both systems after the application of any given number of applications of *flip*. The variable *extraneous* in the second definition is to all intents and purposes invisible externally.

But if we work out the denotation of each system in the semantics given in section 3, we have for the first case:

$$<[1 \leftarrow < [on \leftarrow \lambda\sigma.\ \sigma < 1, on >], [flip \leftarrow \lambda\sigma.\ \sigma[< 1, on > \leftarrow not(\sigma < 1, on >)]] >], \\ [< 1, on > \leftarrow \mathrm{ff}] >$$

But for the second case:

$$<[1 \leftarrow <[on \leftarrow \lambda\sigma.\ \sigma < 1, on >], \\ [flip \leftarrow \lambda\sigma.\ \sigma[< 1, on > \leftarrow not(\sigma < 1, on >), < 1, extraneous > \leftarrow \mathrm{tt}]] >], \\ [< 1, on > \leftarrow \mathrm{ff}, < 1, extraneous > \leftarrow \mathrm{ff}] >$$

These quantities are not equal. Here is a case where two objects with the same externally observable behaviour have different denotations. So the semantics is not fully abstract, according to our definition.

5 A DERIVED FULLY ABSTRACT SEMANTICS

We have shown that Reddy's semantics fails to be fully abstract. We can use it, however, to derive a form of fully abstract semantics. To do this, we employ the "final algebra" construction widely discussed in the theory algebraic specifications [Wan79]. We will use some very elementary concepts from universal algebra [Coh85]. Lack of space means we must omit the proofs of propositions.

In the following, assume S is a system, and that $< \eta, \sigma >$ is its denotation in the first semantics. Let n be the index of the last object declared in S. Let $vars(i, S)$ be the names of variables declared for object $i \leq n$ in S, $exs(i, S)$ its extractors, and $procs(i, S)$ the procedures.

Define a set of *terms*, T_S, the smallest set satisfying:

1. $\bullet \in T_S$,
2. $p\,i\,t \in T_S$, for $t \in T_S$, $1 \leq i \leq n$, $p \in procs(i, S)$.

Next define a function ev: $T_S \rightarrow State$:

1. $\mathrm{ev}\,\bullet = \sigma$,
2. $\mathrm{ev}\,p\,i\,t = \pi_2(\eta\,i)p\,(\mathrm{ev}\,t)$.

Define a congruence $\equiv_{\mathbf{T}}$ on T_S:

$$t \equiv_{\mathbf{T}} t' \text{ iff } \begin{array}{ll} \pi_1(\eta\, i)\, e\, (\mathrm{ev}\, t) = \pi_1(\eta\, i)\, e\, (\mathrm{ev}\, t'), & \text{for } 1 \leq i \leq n,\ e \in obs(i, S), \text{ and} \\ p\, i\, t \equiv_{\mathbf{T}} p\, i\, t', & \text{for } 1 \leq i \leq n,\ p \in procs(i, S). \end{array}$$

Let $[t]$ denote the equivalence class of term t wrt. $\equiv_{\mathbf{T}}$.

Write out a heterogeneous signature Sig_S as follows:[8]

sorts

$Bool, P^1, \ldots, P^n, E^1, \ldots, E^n, State.$

operators

$e : E^i$, for $1 \leq i \leq n$, $e \in exs(i, S)$,
$p : P^i$, for $1 \leq i \leq n$, $p \in procs(i, S)$,

$\nu : State$,
$\beta^1 : E^1 \times State \to Bool, \ldots, \beta^n : E^n \times State \to Bool$,
$\phi^1 : P^1 \times State \to State, \ldots, \phi^n : P^n \times State \to State.$

Next, form a Sig_S-algebra, $\mathbf{T}_S$, such that we have:

carriers

$Bool_{\mathbf{T}_S} = \{\mathrm{tt}, \mathrm{ff}\}$,
$E^1_{\mathbf{T}_S} = exs(1, S), \ldots, E^n_{\mathbf{T}_S} = exs(n, S)$,
$P^1_{\mathbf{T}_S} = procs(1, S), \ldots, P^n_{\mathbf{T}_S} = procs(n, S)$,
$State_{\mathbf{T}_S} = T_S / \equiv_{\mathbf{T}}$.

operations

$e_{\mathbf{T}_S} = e$, for $e \in exs(i, S)$, $1 \leq i \leq n$,
$p_{\mathbf{T}_S} = p$, for $p \in procs(i, S)$, $1 \leq i \leq n$,

$\nu_{\mathbf{T}_S} = [\bullet]$,
$\beta^1_{\mathbf{T}_S} = \lambda e, [t].\pi_1(\eta\, 1)\, e\, (\mathrm{ev}\, t), \ldots, \beta^n_{\mathbf{T}_S} = \lambda e, [t].\pi_1(\eta\, n)\, e\, (\mathrm{ev}\, t)$,
$\phi^1_{\mathbf{T}_S} = \lambda p, [t].[p\, 1\, t], \ldots, \phi^n_{\mathbf{T}_S} = \lambda p, [t].[p\, n\, t]$.

It is not difficult to demonstrate that $\mathbf{T}_S$ is well-defined, and an induction on observations produces the following:

[8] Note that P^n is simply the name of a set, and does not denote the n-th cartesian power of P^1. The use of such superscripts will make our notation clearer later on.

Proposition 1 *Let S and S' be two systems defining objects indexed $1 \ldots n$. Then $S \equiv_{\text{obs}} S'$ iff $\mathbf{T}_S \cong \mathbf{T}_{S'}$, where the algebra-isomorphism $\cong$ is the identity on all sorts except State.*

6 A NATURAL FULLY ABSTRACT SEMANTICS

From proposition 1, we might conclude that our quest for a fully abstract semantics is complete; to compute the fully abstract denotation of a system S, one could simply work out its denotation according to the semantics of section 3, and use it to construct the algebra $\mathbf{T}_S$ as above.

There are two main objections to this approach:

- By this method, two systems which were behaviourally equivalent would only get *isomorphic* denotations (this is all that is guaranteed by proposition 1). Although some branches of mathematics used in programming language semantics usually characterize things only up to isomorphism, it would be nice to give them *equal* meanings.
- Many researchers are unhappy with so-called "denotational" semantics which result from taking quotients of terms, as we do in the construction of $\mathbf{T}_S$. In the context of programming languages, such models are considered to be insufficiently "natural" to be generally useful in reasoning about programs [BCL85].

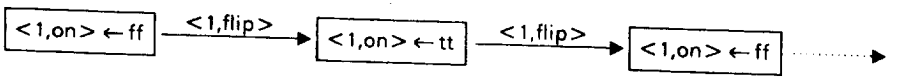
In this section, we will produce a fully abstract semantics which answers these objections. We will not abandon the construction of $\mathbf{T}_S$ entirely, however, since we can use it to show that our new semantics is indeed fully abstract. The techniques we use in this section revolve around the construction of state-transition graphs for systems, and have their roots in classical automata theory [Kam83].

Consider using a tree to represent the states of a system. Each node in this tree will correspond to possible configurations of the objects in the system, and each edge will stand for the execution of a procedure in an object. We will label nodes with functions which map pairs, consisting of the index of an object and an extractor on that object, to the value returned when that extractor is applied to that object in the state represented by the node.

Emerging from each node will be a set of edges, and each such edge will be labelled with another pair—the index of an object, and the name of a procedure on that object. At the other end of that edge will be the node standing for the state of the system which results from applying the procedure to the object in the state at the first node.

For example, from the first definition of the switch system, we derive a very

simple tree which looks (near the root) like:



A vital observation to make is that *exactly the same* tree would result from applying the same method to the second definition of the switch system (the one with the variable *extraneous*). In other words, these trees are *representations of the external behaviour of modules.* This is the basis for the abstract semantics.

In what follows, we will make use of finite sequences or strings over a set of elements. We will use the symbol $\bar{\omega}$ to denote an empty string, and the operator '.' to concatenate two strings. We will sometimes implicitly regard an element as its corresponding one-element string.

We can represent our trees mathematically as collections of functions which map finite strings of edge-labels to node-labels. As we apply procedures to objects in the system, the state of the system will "move" from one node in the tree to the next. We will have one function in the collection for each such position of the state.

To return to our example, the state at the root of the tree would correspond to the function:

$$[\bar{\omega} \leftarrow [< 1, on > \leftarrow \mathrm{ff}],$$
$$< 1, flip > \leftarrow [< 1, on > \leftarrow \mathrm{tt}],$$
$$< 1, flip > . < 1, flip > \leftarrow [< 1, on > \leftarrow \mathrm{ff}], \ldots]$$

For the second node, we would have:

$$[\bar{\omega} \leftarrow [< 1, on > \leftarrow \mathrm{tt}],$$
$$< 1, flip > \leftarrow [< 1, on > \leftarrow \mathrm{ff}],$$
$$< 1, flip > . < 1, flip > \leftarrow [< 1, on > \leftarrow \mathrm{tt}], \ldots]$$

This is just the function for the first node "shifted along by 1." By shifting repeatedly in this fashion, we can generate a function for each node in the tree.

We will use algebras, as in the last section, as the denotations of systems in the abstract semantics. As before, let Sig_S be the signature corresponding to a system S. Let $Sig(i)$ be set of all such signatures for systems defining i objects. A $Sig(i)$-algebra is *standard* if its carrier for the sort *Bool* is the set $\{\mathrm{tt}, \mathrm{ff}\}$, if the carriers of sorts $P^1, \ldots P^i$ and $B^1, \ldots, B^i$ are finite subsets of the sets Id_p and Id_v respectively, and its carrier for *State* is countable. Let $\mathbf{A}_0$ be the standard $Sig(0)$-algebra such that $State_{\mathbf{A}_0} = \{\mathbf{error}\}$ and $\nu_{\mathbf{A}_0} = \mathbf{error}$. Let $\mathcal{A}$ be the union for finite i of the sets of standard $Sig(i)$-algebras.

Now we can describe the sets and functions defining the semantics. We have changed some of the syntactic domains in minor respects to make the second semantics easier to write.

Semantic Domains

$\epsilon \in VMap = Id_v \to \mathbf{B}$
$\mathbf{A} \in \mathcal{A}$
$\tau \in State_{\mathbf{A}}$

Semantic Functions

$P' : program \to \mathcal{A}$
$O' : object\text{-}definition \to \mathbf{N} \to \mathcal{A} \to \mathcal{A}$
$S' : stmt \to \mathbf{N} \to \mathcal{A} \to (VMap \times State_{\mathbf{A}}) \to (VMap \times State_{\mathbf{A}})$

Semantic Equations

$P'[\![\textbf{object } 1 \textbf{ has } object\text{-}definition_1, \ldots, \textbf{object } \bar{n} \textbf{ has } object\text{-}definition_n]\!] =$
$\quad O'[\![object\text{-}definition_n]\!]\, n(\ldots O'[\![object\text{-}definition_1]\!]\, 1\, \mathbf{A}_0 \ldots)$

$O'[\![\textbf{variables } v_1 = \bar{b}_1, \ldots, v_k = \bar{b}_k$
$\quad \textbf{extractors } e_1, \ldots, e_l$
$\quad \textbf{procedures } p_1 \textbf{ is } stmt_1, \ldots, p_m \textbf{ is } stmt_m]\!]\, i\mathbf{A} = \mathbf{A}'$ (defined below)

$S'[\![stmt_1; stmt_2]\!]\, i\mathbf{A} < \epsilon, \tau > = S'[\![stmt_2]\!]\, i\mathbf{A}(S'[\![stmt_1]\!]\, i\mathbf{A} < \epsilon, \tau >)$
$S'[\![v \leftarrow v']\!]\, i\mathbf{A} < \epsilon, \tau > = < \epsilon[v \leftarrow \epsilon v'], \tau >$
$S'[\![v \leftarrow \vec{j}.e]\!]\, i\mathbf{A} < \epsilon, \tau > = < \epsilon[v \leftarrow \beta^j_{\mathbf{A}}(e, \tau)], \tau >$
$S'[\![\vec{j}.p]\!]\, i\mathbf{A} < \epsilon, \tau > = < \epsilon, \phi^j_{\mathbf{A}}(p, \tau) >$

We define the algebra $\mathbf{A}'$ in the following manner:[9]

Let P^* be the least set satisfying:

1. $\bar{\omega} \in P^*$,
2. $\bar{p}. < j, p > \in P^*$, for $\bar{p} \in P^*$, $1 \le j < i$, $p \in P^j_{\mathbf{A}}$,
3. $\bar{p}. < i, p_j > \in P^*$, for $\bar{p} \in P^*$, $1 \le j \le m$.

$\mu : P^* \to (VMap \times State_{\mathbf{A}})$ be a function defined inductively as follows:

$$\mu(\bar{\omega}) = < [v_1 \leftarrow b_1, \ldots, v_k \leftarrow b_k], \nu_{\mathbf{A}} >,$$

$$\mu(\bar{p}. < j, p >) = < \pi_1(\mu(\bar{p})), \phi^j_{\mathbf{A}}(p, \pi_2(\mu(\bar{p}))) >, \quad \text{for } 1 \le j < i,\ p \in P^j_{\mathbf{A}},$$

$$\mu(\bar{p}. < i, p_j >) = S'[\![stmt_j]\!]\, i\mathbf{A}\mu(\bar{p}), \quad \text{for } 1 \le j < m.$$

[9] The index i, algebra $\mathbf{A}$ and such like are assumed to be those in the semantic equation defining O'.

Let $\alpha : P^* \to (\{e_1, \ldots, e_l\} \to \mathbf{B})$ be defined by

$$\alpha(\bar{p}) = [< j, e > \leftarrow \beta^j_{\mathbf{A}}(e, \pi_2(\mu(\bar{p}))),\ < i, e_h > \leftarrow \pi_1(\mu(\bar{p}))e_h], \quad \text{for } 1 \le j < i,\ e \in E^j_{\mathbf{A}},\ 1 \le h \le l.$$

Now define the carriers of $\mathbf{A}'$ as follows:

$Bool_{\mathbf{A}'} = \{\mathrm{tt}, \mathrm{ff}\}$,
$E^1_{\mathbf{A}'} = E^1_{\mathbf{A}}, \ldots, E^{i-1}_{\mathbf{A}'} = E^{i-1}_{\mathbf{A}}$,
$P^1_{\mathbf{A}'} = P^1_{\mathbf{A}}, \ldots, P^{i-1}_{\mathbf{A}'} = P^{i-1}_{\mathbf{A}}$,
$E^i_{\mathbf{A}'} = \{e_1, \ldots, e_k\}$,
$P^i_{\mathbf{A}'} = \{p_1, \ldots, p_m\}$,

$State_{\mathbf{A}'} = \{\lambda \bar{p}' \in P^*.\alpha(\bar{p}.\bar{p}') \mid \bar{p} \in P^*\}$.

The operations of $\mathbf{A}'$: First, the operations to name extractors and procedures:

$e_{\mathbf{A}'} = e$, for $e \in E^j_{\mathbf{A}'}$, $1 \le j \le i$,
$p_{\mathbf{A}'} = p$, for $p \in P^j_{\mathbf{A}'}$, $1 \le j \le i$.

To produce the initial state of the system:

$\nu_{\mathbf{A}'} = \lambda \bar{p}.\alpha(\bar{p})$.

To apply extractors and procedures:

$\beta^j_{\mathbf{A}'} = \lambda e, \tau.\tau\bar{\omega} < j, e >$, for $1 \le j \le i$,
$\phi^j_{\mathbf{A}'} = \lambda p, \tau.(\lambda \bar{p} \in P^*.\alpha(< j, p > .\bar{p}))$, for $1 \le j \le i$.

In order to see informally how the semantics works, let us look at the semantic equations. The first is straightforward, building the denotation of a program (or a system) from a sequence of object-definitions.

The second is more complex. In it, we take an object-definition, together with the denotation A of a system, and produce a new algebra, $\mathbf{A}'$. This is the denotation of a system obtained by appending the object-definition to the old system. We derive $\mathbf{A}'$ in three steps:

The first involves generating all possible configurations of both the new object and of the system on which it is defined. This is done using the function μ, which maps sequences of edge-labels to pairs. The first component of such a pair is a function recording the values of the variables in the new object, and the second component is a state of the system. The pair $\mu(\bar{\omega})$ has all the variables in the new object set to their initial values, and the system in its initial state. To move to a new configuration, we apply a procedure to an object indexed $1, \ldots, i$. When applying a procedure to objects $1, \ldots, i-1$, we leave the variables in object i unchanged

(since a procedure cannot cause the variables of an object defined later than it to change), and compute the new state of the system using operations in A. To apply a procedure to object i, we execute its definition using the semantic function S'.

The second stage of the process defines a function α, which computes the node-labels in the tree for the new system, by sampling the values of extracted variables as computed by μ.

Finally, we can write out the definition of $\mathbf{A}'$ directly. Operations in $\mathbf{A}'$ which apply extractors merely "pull out" the first node label from a given state, and apply it directly to an object-number/extractor-name pair. Procedure-applying operations select a new function from $State_{\mathbf{A}'}$ which represents the new state of the prelude, using the "shifting" trick illustrated in the example.

It is easy to show that $\mathbf{A}'$ is an algebra in $\mathcal{A}$. Furthermore, we have finally found what we are looking for:

Proposition 2 *For a system S, let $\mathbf{T}_S$ be the algebra representing S as derived in section 5. Let $\mathbf{A}_S$ be the corresponding algebra constructed using the abstract semantics given above. Then $\mathbf{A}_S \cong \mathbf{T}_S$, where $\cong$ is the identity on all sorts except State.*

Proposition 3 *For two algebras $\mathbf{A}_S$ and $\mathbf{A}_{S'}$ which are denotations of systems according to the abstract semantics, if $\mathbf{A}_S \cong \mathbf{A}_{S'}$, (where $\cong$ is the identity on all sorts except State), then $\mathbf{A}_S = \mathbf{A}_{S'}$.*

So, by proposition 1, we have:

Corollary 1 *In the abstract semantics, for two systems S_1 and S_2*

$$P'[\![S_1]\!] = P'[\![S_2]\!] \text{ iff } S_1 \equiv_{\text{obs}} S_2$$

7 CONCLUSION

The corollary tells us that we have developed a fully abstract semantics, in the sense that two systems with the same observable behaviour will be given equal denotations. This has not been achieved in any previous semantics for object-oriented languages. However, much remains to be done.

As we pointed out in the introduction, the language we used here is one of the most basic object-oriented languages imaginable. Extending the treatment to account for more complex notations is an obvious next step. We end with some comments on such extensions:

- Allowing conditional and iterative constructs in statements (to make the language computationally complete) is easy. One would simply make the carrier

of sort *State* in the denotation of a system into a flat (Scott) domain. The semantic function S' would then be extended in the usual manner. This would also allow one to account for partial operations and errors.

- Parameter-passing does not appear too difficult to describe. Certainly, passing parameters of "base" type (booleans in our case) can be accomodated easily, using techniques like those used for CCS [Mil80]. Passing reference parameters does not seem to be much more difficult—but see the points below.
- Giving full abstract denotations to a system (i.e. a collection of objects) with cyclic chains of references between objects should not be too hard, *provided all the objects involved in cycles of reference occur in the system being described.* We will see the reason for this caveat presently.
- Dynamic object-creation could be tackled using a form of "sharing semantics" like that in [Bro85].

Another direction in which we might take our investigation involves Stoughton's concept of *contextual full abstraction.* In the semantics in this paper, we gave full abstract denotations to systems only. A contextually fully abstract semantics would also give full abstract denotations to object-definitions.

In a language like ours, which disallows cyclic object-dependencies, this should not be too onerous. The main problem lies in defining the semantic domains for systems in a sufficiently precise fashion, so that all of their members may be produced as the denotations of some system definition in the language.[10] Briefly, one could define a succession of types of denotation, $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots$, corresponding to systems with $0, 1, 2, \ldots$ objects. The meaning of an object-definition describing object i would then be a function from denotations of type $\mathcal{A}_{i-1}$ to ones of type $\mathcal{A}_i$.

Unfortunately, once we allow cyclic chains of reference among objects, the contextual full abstraction problem becomes much more complicated. In this case, one would need to represent object-definitions as members of recursively-defined functional types. Producing "natural" fully abstract semantics for such functions is known to be very difficult (at least in the case of sequential languages like the one in this paper) [BCL85]. As far as we know, at the time of writing, no wholly satisfactory way of producing such a semantics exists.

Though we have good reason to believe that the speculations in this section are correct, they have not been verified in detail. Work is underway at the moment to substantiate the above suppositions, and we hope to report on progress in the near future.

The presentation here has been one of "small beginnings." We would like to believe that we have delineated some salient aspects of the semantics of object-oriented languages, and that we have provided at least a modest foundation for future developments in the field.

[10] The definition of $\mathcal{A}$ used in this paper is intended only to ensure a) that the operations carried out on $\mathbf{A}$ in defining $\mathbf{A}'$ are defined, and b) that $\mathcal{A}$ is a well-defined in classical set theory.

Acknowlegements I would like to acknowledge the work of Mario Wolczko, Samuel Kamin and Uday Reddy in providing an invaluable basis for this investigation. I am most grateful to Mario Wolczko in particular for his guidance and encouragement. I would like to thank Alan Mycroft and all those in Cambridge who have been so fullsome in their support. My thanks go also to the referees, for their careful and cogent comments on the first version of this paper.

References

[ADKR86] P. America, J.W. DeBakker, J.N. Kok, and J.J.M.M Rutten. *A Denotational Semantics for a Parallel Object-Oriented Language.* Technical Report, Center for Mathematics and Computer Science, Amsterdam, 1986.

[BCL85] G. Berry, P.-L. Curien, and J.-J. Levy. Full abstraction for sequential programming languages: the state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, Cambridge University Press, 1985.

[Bro85] S. Brookes. A fully abstract semantics and a proof system for an ALGOL-like programming language with sharing. In *Conference on Mathematical Foundations of Programming Semantics*, 1985.

[Coh85] P. Cohn. *Universal Algebra.* Wiley, 1985.

[Cox83] B. J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1983.

[GM87] J.A. Goguen and J. Meseguer. *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics.* Technical Report, Center for the Study of Language and Information, 1987.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hen88] M. Hennessy. *The Algebraic Theory of Processes.* M.I.T. Press, 1988.

[Kam83] S. Kamin. Final data types and their specification. *A.C.M. TOPLAS*, 5(1), 1983.

[Kam88] S. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *A.C.M. Symp. On Principles of Programming Languages*, 1988.

[Mey88] A. Meyer. Semantical paradigms: notes for an invited lecture. In *Symposium on Logic in Computer Science*, 1988.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[Plo77] G. Plotkin. L.C.F. considered as a programming language. *Theoretical Computer Science*, 5, 1977.

[Red88] U. S. Reddy. Objects as closures: abstract semantics of object-oriented languages. In *Conference on LISP and Functional Programming*, 1988.

[Ren82] T. Rentsch. Object-oriented programming. *A.C.M. SIGPLAN*, 17(9), 1982.

[Sto88] A. Stoughton. *Fully Abstract Models of Programming Languages*. Pitman, 1988.

[Wan79] M. Wand. Final algebra semantics and data-type extensions. *Journal of Computer and System Sciences*, 19, 1979.

[Wol87] M. Wolczko. Semantics of Smalltalk-80. In J. Bezivin, J. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the 1987 European Conference on Object-Oriented Programming*, Springer-Verlag, 1987.

[Wol88] M. Wolczko. *Semantics of Object-Oriented Languages*. PhD thesis, University of Manchester, 1988.