Game semantics for object-oriented languages: a progress report*

John Longley Nicholas Wolverson

July 25, 2006

Abstract

We report on ongoing work in the area of game semantics for object-oriented languages. We present a simple language that embodies many of the core ingredients of object-oriented languages, focusing especially on the issues of data abstraction, inheritance, and shared mutable state. We give a denotational semantics for this language using a particularly simple category of games, and describe some of the seemingly novel ideas that are involved in the proof of soundness for this semantics. We also indicate how various possible extensions of our language might be modelled, and discuss the place of the present work within our long-term program of using game semantics to inform the design of object-oriented languages and type systems for them.

1 Introduction

In this paper we report on an ongoing project concerned with applying ideas from game semantics to the study of object-oriented languages [Lona]. The spirit of this project is to work with a very simple category of games with a rich mathematical structure and see what kinds of object-oriented languages can be naturally modelled within it, rather than (say) trying to model every aspect of existing object-oriented languages by developing suitably elaborate mathematical machinery. Our expectation is that the simplicity and naturalness of our underlying game model will lead us to identify fragments of existing languages that are relatively well-behaved and mathematically pleasant. This can be expected to lead to relatively clean and tractable principles for reasoning about programs in such fragments, as well as statically enforceable constraints on programs (such as might be incorporated into a type system for a language) which guarantee that they fall within these well-behaved fragments. A longer term, more ambitious goal is to allow the game model to guide the design of a *full-scale* object-oriented language with a sound mathematical basis, possibly incorporating novel language features inspired by the mathematical structures that are available in the model.

In this paper, we make a modest start on our programme by presenting a simple-minded game semantics for a language that embodies many, but not all,

*Research supported by EPSRC Grant GR/T08791: "A programming language based on game semantics".

of the core ingredients of object-oriented languages. Let us briefly discuss these ingredients in turn.

Firstly, we wish to take seriously the issue of *data abstraction*. For simplicity, imagine a Java-style object whose fields are all private and whose methods are all public. The basic idea behind our semantics is that the denotation of such an object should be a strategy specifying the externally observable behaviour of the object under all possible sequences of future interactions via method calls. In other words, the object is considered as a black box that one can interact with by sending messages to it, or more specifically by playing moves in an appropriate game — all *internal* interactions involving the contents of the fields are hidden when we pass to the denotation of the object itself. The *type* of the object (corresponding in Java terms to the interface it implements) will determine what *kinds* of interactions are legitimate, and will thus define a game whose strategies represent possible behaviours for objects of this type.

In view of this emphasis on externally visible behaviour, it is to be expected that this approach should lead naturally to a fully abstract semantics for objects, in the sense that observationally equivalent objects are identified, even if they arise from different classes that implement the same interface. In view of the recognised importance of this kind of data abstraction within software engineering, it is perhaps surprising that relatively little attention has hitherto been devoted to the problem of providing fully abstract denotational semantics for abstract datatypes. Our work here shows that game semantics is particularly well suited to this task.

A second aspect of object-oriented languages which we wish to focus on is the area of *inheritance*, including Java-style method overriding and dynamic binding. As we shall see, these can be accommodated fairly easily in our framework by taking a certain view of what the denotation of a class body should be. Interestingly, it is the extension of classes with additional *fields* (rather than methods) that requires the most work (see Section 7.1).

Thirdly, there is the issue of *mutable state* and the resulting phenomenon of *interference*. In all object-oriented languages, objects can share state in complex ways, so that interactions with one object can indirectly affect or interfere with the behaviour of another. One important aspect of our semantics is the way in which such effects are handled correctly in the presence of the complicated kinds of flow of control that can arise from the execution programs. However, as regards the question of the kinds of state-changing operations that methods can perform, we are at present adopting a middle ground between only considering ground-type state on the one hand, and considering unrestricted state operations on the other. This means that whilst our language is rich enough to allow us to explore some interesting phenomena, it does not cover all possible uses of state that might arise in a language such as Java. It seems likely that more general uses of state could be accommodated within our semantic framework, but at the cost of some further complication.

A fourth issue concerns the identity and newness of *names* (which typically appear in object-oriented languages as *references*). This issue is the focus of current work on game models based on nominal sets (e.g. [AGM$^+$04]). In the present paper, we do not attempt to address this issue at all. This is, in part, a consequence of our decision to focus initially on those language features that our model naturally seems to support, and in particular our decision to view objects purely in terms of their external behaviour. However, we do also have in mind an approach to modelling names within our present semantic framework, at the cost of a little more machinery. We hope the details will appear elsewhere.

Inevitably, our decision not to consider names places restrictions on the language we are able to model. Most obviously, we are unable to deal with equality tests for references (== in Java). However, the following example shows that the notion of identity of references is sometimes needed even when describing constructs that do not explicitly refer to it. Consider a recursively defined class C with just two fields, a field a of integer type and a field c of type C (in Java terminology). Construct an object o_1 with $o_1.a = 0$, then construct objects o_2, o_3 with $o_2.a = o_3.a = 1$ and $o_2.c = o_3.c = o_1$. Because o_2 and o_3 have indistinguishable structure and are only distinguished by their references, they will be assigned the same denotation by any semantics that does not take account of references. However, it would seem that such a semantics will then be unable to account for the fact that assigning $o_2.c = o_2$ is observably different in its effect from $o_2.c = o_3$ (since the value of $o_2.c.c.a$ after the assignment will be 1 in the former case and 0 in the latter case). This example suggests that if we seriously wish to model a realistic object oriented language we will need to take account of identity of objects sooner or later, even if we do not care about explicit reference equality. (This argument is somewhat misleading in that a similar argument might appear to suggest that *all* forms of aliasing are problematic for our approach. In fact this is not the case; the problem is quite specific to the presence of cycles in the heap.)

For the present, however, our focus is on exploring how far one can get by treating objects purely in terms of their abstract behaviours, and indeed, on clarifying which kinds of computation with objects really involve only their abstract behaviour, and which kinds make essential use of the references to them as well. (One could imagine, for instance, a language with a type system that allowed one to express this distinction, and this might be helpful for reasoning about programs.) For example, our language will be restricted in such a way that only *acyclic* heaps can be constructed — this will rule out the example given above, which clearly depends on the possibility of cycles in the heap.

Finally, we should remark that we are only attempting to model *sequential* (as opposed to concurrent) programming languages; indeed, sequentiality would seem to be built into the fabric of the game model we are using. However, our semantic framework does offer support for *coroutining*, which can be viewed as a "sequential" form of threading in which the passage between threads is entirely under the programmer's control. There is therefore the possibility of extending our approach to languages with this kind of feature.

1.1 An alternative approach: ML with references

Many of the ingredients required for the construction of an object-oriented language, such as the key idea of *local state*, have already been treated by previous work in game semantics: indeed, game models for languages with ML-style references (including higher-order state) are given in [AHM98], [Lai02]. Moreover, it is shown in [BPSM99] how a core object-oriented language may be translated into such a language, so from one point of view, it is in essence already known how to give game models for object-oriented languages.

However, we believe that our more direct approach is of interest for several reasons. Firstly, it is a core element of our long-term methodology that our choice of programming language should be guided by simple and beautiful mathematical structures, and this would be lost if instead of working directly with our game models we allowed our choice of language to be unduly influenced by what happened to be (straightforwardly) expressible in any particular target language. Secondly, in our view our approach offers a somewhat deeper analysis of the notion of state: rather than take as given the behaviour of reference cells considered as "objects" with read and write methods, we are able to give an explanation of where this abstract behaviour comes from in terms of the underlying concrete state. Thirdly, giving a semantics via an intermediate language tends to obscure issues of full abstraction: even if our semantics for that language is fully abstract, there remains the question of whether all observations on programs that can be performed in the intermediate language also have a counterpart in the source language. Fourthly, the semantics we give is mathematically different from the one obtained by this indirect route, and is in some ways more immediately adapted to the specific phenomena associated with object-oriented languages; this in itself gives our approach some independent interest. More detailed comparison of these alternative approaches at a later date would be useful.

1.2 Structure of paper

The remainder of the paper is structured as follows.

In Section 2 we introduce the syntax and typing rules for the object-oriented language we will be working with. Like Java, our language is class-based and method invocation is call-by-value. The core of our language is in part inspired by [BPSM99]; like many languages intended for theoretical study, it has a somewhat functional flavour, and is also more consistently "higher-order" in character than most existing object-oriented languages. Though its syntax may look quite unlike that of real-world languages, it is intended as a suitable target language for translations from Java-like languages, and we illustrate this by providing examples of derived constructs employing more Java-like syntax.

In Section 3 we present a straightforward operational semantics for our language using a simple model for heaps. We hope that it will be more or less directly apparent that this does indeed capture the kind of operational behaviour familiar from Java; this contention is reinforced by the close resemblance between our operational semantics and attempts by other workers to formalise the semantics of portions of Java.

In Section 4 we introduce the category of games we will work with, essentially the one introduced by Lamarche [Lam92]. The games we work with are very simple (there is no need for justification pointers, for instance), but they lead to an extremely rich structure. Our approach is somewhat in the spirit of [AJM94], in that we represent multiple uses of an object by means of a linear exponential giving us an infinite supply of "copies" of the object's behaviour. The essential difference is that in [AJM94] only history-free strategies were considered, with the effect that all these copies behaved identically, whereas in our setting it is crucial that the copies can interfere with one another, in view of the presence of mutable state. Having constructed our base category $\mathscr{C}$ of games, we then pass to a call-by-value category $\mathscr{D}$ by means of a simplified version of the $\mathbf{Fam}(\mathscr{C})$ construction of [AM98].

In Section 5, we present a compositional denotational interpretation of our language in the category $\mathscr{D}$. The main technical task here is the construction of a certain morphism called *thread*, which allows us to pass from a "concrete" view of an object, in which the internal state and interactions with it are exposed, to an "abstract" view in which these internal interactions are hidden and only the externally visible interactions are retained.

Our operational and denotational semantics are markedly different in many ways — for instance, the denotational semantics makes no reference at all to heaps. As a consequence, even the proof of soundness (usually the "easy" half of adequacy) turns out to be far from straightforward. In Section 6 we give a brief account of the proof, concentrating on some of the interesting and seemingly new ingredients that are required. Proofs of completeness (the other half of adequacy), along with the expected full abstraction and universality properties, are the focus of continuing work.

In Section 7 we briefly discuss some of the extensions and variants of our language that we have considered. For some of these, a minor elaboration of our existing semantics is all that is required, whilst others seem to require deeper innovations and are the subject of ongoing investigation.

2 Language

We shall work with a linear (or affine) call-by-value lambda calculus, where $A \multimap B$ is the type of functions which "consume their argument" of type A to produce a result of type B. We make contraction available only for certain *reusable* types (see the predicate $re(-)$ defined in Figure 1).

We shall view an *object* as a collection of methods which may be invoked repeatedly with some argument, approximately a reusable record of functions which may behave in a stateful manner. The calculus shall be class-based, so we take objects to be created from *classes* via the **new** operator. We consider classes with a single updatable field, since we can consider multiple fields to be a single field of tuple type. On the other hand, we explicitly consider multiple methods, partly because method names play a role in overriding. In Java terminology, we consider all fields to be `protected` and all methods `public`; public fields or private methods can easily be simulated. We shall not consider Java-style constructors, but instead implicitly always take the constructor which initialises all fields to the provided values—but handling the general case should not be problematic.

As in [BPSM99], we take classes to be first-class expressions rather than a top level construct, for simplicity and with a view to defining a language with a higher-order flavour (perhaps as an extension of that described here). We create classes via a general class extension mechanism **extend** e **with** e' where one can extend a base class to define a class directly.

To define a class, one must give a collection of named method implementations in a fashion allowing for recursion. A key principle of object-oriented programming is that of open recursion, via method overriding. Methods are defined in a context with a *self* object, standing for an instance of the class presently being defined. Recursive method invocations via *self* refer not to their currently defined implementations, but to the potentially redefined implementations in a future subclass. We thus define a class with an expression

$$\textbf{extend}\ e\ \textbf{with}\ (\varsigma)\ \{m_1 = e_1, \ldots, m_n = e_n\}$$

as a collection of functions $e_1 \dots e_n$ labelled with method names $m_1 \dots m_n$, where ς is the *self*-binding, and then might use it an expression such as

$$(\mathbf{new}\ (\mathbf{extend}\ e\ \mathbf{with}\ \dots)\ s).m\ e'$$

where we are subclassing e, creating a new object with state s, and then invoking method m with argument e'.

As classes define stateful objects, we choose to take a method implementation to be a state transforming function of type

$$m\colon S \otimes X \to S \otimes Y$$

for a class with state of type S, where we are defining a method m of type $X \to Y$. As we will discuss later, this will be problematic for general S, and in fact we take

$$m\colon (O \otimes T) \otimes X \to T \otimes Y$$

replacing the state S with a ground type component T and an object type component O. The lack of an O on the right hand side does not disallow interaction with a (stateful) object stored in the state, but effectively disallows replacing this with a new object.

This interpretation can be considered as a first step to a more general treatment of objects. Viewing a method as a function taking a state as argument and returning a modified state can be thought of as allowing the state to be read at the start of a method's execution (taking a private copy) and written at the end. We might wish to extend this to allow the state to be read from or written to at arbitrary points. However, one can simulate this behaviour by wrapping the ground-type state in an object, with methods which can be used to read or write this state at any time.

Base Calculus

While the constructs presented above represent our conception of classes, we shall actually regard these as derived from more basic operations on objects. We could define a language with built-in classes, but interpretation of these in our operational and denotational semantics would involve a duplication of effort (and require longer proofs). For our operational semantics, we must split class instantiation into two steps in any case, so we work in a simpler setting with objects, a fixed point operator and a state-internalisation operation, which is sufficient to implement classes as described above.

Firstly, we have objects which can be created directly:

$$\mathbf{obj}\ \{m_1 = e_1, \dots, m_n = e_n\}$$

These are reusable, and hence must be defined in a reusable context, that is one consisting entirely of objects. We freely use a shorthand

$$\mathbf{obj}\ \{m_i = e_i\}_{i \in X}$$

for a set X of indices, treating objects as unordered justified by our subtyping relation.

$$\begin{array}{rcl}\tau, \sigma & ::= & N \mid \tau_1 \otimes \tau_2 \mid \tau_1 \to \tau_2 \mid \\ & & \mathbf{Obj}\ \{m_1\colon \tau_1, \ldots, m_n\colon \tau_n\}\end{array}$$

$$\begin{gathered} \mathit{basic}(N) \\ \mathit{basic}(X) \wedge \mathit{basic}(Y) \to \mathit{basic}(X \otimes Y) \end{gathered}$$

$$\begin{gathered} \mathit{re}(\mathbf{Obj}\ X) \quad \mathit{basic}(X) \to \mathit{re}(X) \\ \mathit{re}(X) \wedge \mathit{re}(Y) \to \mathit{re}(X \otimes Y) \\ \mathit{re}(x_1\colon \tau_1, \ldots, x_n\colon \tau_n) \leftrightarrow \forall i.\mathit{re}(\tau_i) \end{gathered}$$

Figure 1: Types

We shall interpret a class as its *step function*, which takes an implementation of *self* and returns a refined one

$$\mathbf{Class}\ X = \mathbf{Obj}\ X \to \mathbf{Obj}\ X$$

We shall take the fixed point of this step function to obtain the resulting object, but—corresponding to the late binding of *self*—we leave the step function of a class open. To extend a class, one then provides a new step function from the old, adding or altering fields of the resulting object to add or override methods respectively.

This interpretation of classes as step functions is as used in [BPSM99], and discussed in [AC96]. As noted there, overriding is handled correctly, but method update[1] is not permitted, and field update must be handled separately. We are happy to consider objects as being created from classes, rather than using method update, and wish to explicitly consider object state (i.e. fields) in any case.

The last element required to implement classes internalises the stateful behaviour of an object. Given an explicitly state-transforming object *obj* (as arising from the fixed point of a step function), and an initial state s,

$$\mathbf{constr}\ \mathit{obj}\ s$$

gives an object where the state is hidden, incorporated into the behaviour of the object. The usual **new** operation is then just **constr** combined with the fixed point operator.

Figures 1–5 describe our base calculus, and Figures 6–7 derived syntax and rules for classes. While we use x, y to range over variables, we use l for variables of reusable type. The definition of *values* in Figure 3 is required for the object construction rule.[2]

We feel that the derived nature of our classes makes our language more modular, and will make it easier to study possible language extensions. For example, one can easily extend the class syntax shown to provide a **super** facility, for invoking overridden methods of the superclass, without modifying the core language as presented.

[1] That is, replacing the methods of an existing object.

[2] The restriction to values in this rule permits the cases we are interested in (where v will be of the form $\lambda x.e$) and appears necessary to ensure the correctness of our denotational semantics.

$$\begin{array}{rcl} e & ::= & c_\varphi \mid \textbf{ifz}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid \\ & & \langle e_1, e_2 \rangle \mid \textbf{let}\ \langle x, y \rangle\ \textbf{be}\ e_1\ \textbf{in}\ e_2 \mid \lambda x.e \mid e_1\ e_2 \mid \\ & & \textbf{obj}\ \{m_1 = e_1, \ldots, m_n = e_n\} \mid e.m \mid Y(e) \mid \textbf{constr}\ e_1\ e_2 \end{array}$$

Figure 2: Terms

$$\begin{array}{rcl} v & ::= & l \mid c_\varphi \mid \lambda x.e \mid \langle v_1, v_2 \rangle \mid \textbf{obj}\ \{m_1 = v_1, \ldots, m_n = v_n\} \\ & & Y(v) \mid l.m \mid Y(v).m \end{array}$$

Figure 3: Values

$$\frac{}{x\colon \tau \vdash x\colon \tau} \quad \frac{\Gamma, x\colon \tau_1, y\colon \tau_2, \Delta \vdash e\colon \tau}{\Delta, y\colon \tau_2, x\colon \tau_1, \Gamma \vdash e\colon \tau} \quad \frac{\Gamma \vdash e\colon \tau}{\Gamma, x\colon \tau_1 \vdash e\colon \tau}$$

$$\frac{}{\vdash c_\varphi\colon N \otimes \ldots \otimes N \to N}\ \varphi\colon \mathbb{N}^k \rightharpoonup \mathbb{N}, k \geq 0$$

$$\frac{\Gamma \vdash e\colon N \quad \Delta \vdash e_1\colon \tau \quad \Delta \vdash e_2\colon \tau}{\Gamma, \Delta \vdash \textbf{ifz}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2\colon \tau}$$

$$\frac{\Gamma_1 \vdash e_1\colon \tau_1 \quad \Gamma_2 \vdash e_2\colon \tau_2}{\Gamma_1, \Gamma_2 \vdash \langle e_1, e_2 \rangle\colon \tau_1 \otimes \tau_2} \quad \frac{\Gamma, x : \tau_1, y : \tau_2 \vdash e : \tau \quad \Delta \vdash p\colon \tau_1 \otimes \tau_2}{\Gamma, \Delta \vdash \textbf{let}\ \langle x, y \rangle\ \textbf{be}\ p\ \textbf{in}\ \ e\colon \tau}$$

$$\frac{\Gamma, x\colon \tau \vdash e\colon \tau'}{\Gamma \vdash \lambda x.e\colon \tau \to \tau'} \quad \frac{\Gamma \vdash e\colon \tau \to \tau' \quad \Delta \vdash e'\colon \tau}{\Gamma, \Delta \vdash e\ e'\colon \tau'}$$

$$\frac{\Gamma \vdash v_1\colon \tau_1 \ \cdots\ \Gamma \vdash v_n\colon \tau_n}{\Gamma \vdash \textbf{obj}\ \{m_1 = v_1, \ldots, m_n = v_n\}\colon \textbf{Obj}\ \{m_1\colon \tau_1, \ldots, m_n\colon \tau_n\}}\ \mathit{re}(\Gamma)$$

$$\frac{\Gamma, x\colon \sigma, y\colon \sigma \vdash e\colon \tau}{\Gamma, z\colon \sigma \vdash e[z/x, z/y]\colon \tau}\ \mathit{re}(\sigma) \quad \frac{\Gamma \vdash e\colon \textbf{Obj}\ \{m\colon \tau\}}{\Gamma \vdash e.m\colon \tau}$$

$$\frac{\Gamma \vdash e\colon (\tau \to \tau) \to (\tau \to \tau)}{\Gamma \vdash Y(e)\colon \tau \to \tau}\ \mathit{re}(\Gamma) \quad \frac{\Gamma \vdash e\colon \textbf{Obj}\ X \to \textbf{Obj}\ X}{\Gamma \vdash Y(e)\colon \textbf{Obj}\ X}\ \mathit{re}(\Gamma)$$

$$\frac{\Gamma \vdash c\colon \textbf{Obj}\ \{m\colon \sigma \otimes \gamma \otimes \tau_m \to \gamma \otimes \tau'_m\}_{m \in A} \quad \Delta \vdash e\colon \sigma \otimes \gamma}{\Gamma, \Delta \vdash \textbf{constr}\ c\ e\colon \textbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m \in A}}\ \mathit{basic}(\gamma), \mathit{re}(\sigma)$$

$$\frac{\Gamma \vdash e\colon \tau}{\Gamma \vdash e\colon \tau'}\ \tau <: \tau'$$

Figure 4: Core Language

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \qquad \frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \otimes \tau_2 <: \tau_1' \otimes \tau_2'} \qquad \frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

$$\frac{}{\mathbf{Obj}\ \{m_1\colon \tau_1, \ldots, m_n\colon \tau_n\} <: \mathbf{Obj}\ \{m_1\colon \tau_1, \ldots, m_{i+1}\colon \tau_{i+1}, m_i\colon \tau_i, \ldots, m_n\colon \tau_n\}}$$

$$\frac{\tau_1 <: \tau_1' \quad \cdots \quad \tau_n <: \tau_n'}{\mathbf{Obj}\ \{m_1\colon \tau_1, \ldots, m_n\colon \tau_n, m_{n+1}\colon \tau_{n+1}, \ldots\} <: \mathbf{Obj}\ \{m_1\colon \tau_1', \ldots, m_n\colon \tau_n'\}}$$

Figure 5: Subtyping

$$\frac{}{\vdash \mathit{Object}_{\sigma,\gamma}\ : \mathbf{Class}\ \langle \sigma, \gamma; \rangle}$$

$$\frac{\begin{array}{l}\Gamma \vdash e\colon \mathbf{Class}\ \langle \sigma, \gamma; m\colon \tau_m \to \tau_m' \rangle_{m \in A} \\ \Gamma, \varsigma\colon \mathbf{Obj}\ \{e_m\colon \sigma \otimes \gamma \otimes \tau_m \to \gamma \otimes \tau_m'\}_{m \in A \cup B} \\ \quad \vdash e_m\colon \sigma \otimes \gamma \otimes \tau_m \to \gamma \otimes \tau_m'\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{extend}\ e\ \mathbf{with}\ (\varsigma)\ \{m = e_m\}_{m \in B} \\ \qquad : \mathbf{Class}\ \{\sigma, \gamma; m\colon \tau_m \to \tau_m'\}_{m \in A \cup B}\end{array}}\ m \in B$$

$$\frac{\Gamma \vdash e_1\colon \mathbf{Class}\ \langle \sigma, \gamma; m\colon \tau_m \to \tau_m' \rangle_{m \in X} \quad \Delta \vdash e_2\colon \sigma \otimes \gamma}{\Gamma, \Delta \vdash \mathbf{new}\ e_1\ e_2\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau_m'\}_{m \in X}}$$

Figure 6: Derived Constructs

$$\begin{array}{c}\mathbf{Class}\ \langle \sigma, \gamma; m\colon \tau_m \to \tau_m' \rangle_{m \in X} \\ \leadsto \\ \mathbf{Obj}\ \{m\colon \sigma \otimes \gamma \otimes \tau_m \to \gamma \otimes \tau_m' \rangle_{m \in X} \quad \to \\ \mathbf{Obj}\ \{m\colon \sigma \otimes \gamma \otimes \tau_m \to \gamma \otimes \tau_m' \rangle_{m \in X}\end{array}$$

$$\mathit{Object}_{\sigma,\gamma} \leadsto \lambda x.\{\}$$

$$\begin{array}{c}\mathbf{extend}\ c\ \mathbf{with}\ (\varsigma)\ \{m = e_m\}_{m \in B} \\ \leadsto \\ \lambda\varsigma.\ \mathbf{obj} \left\{ \begin{array}{ll} m = (c\ \varsigma).m, & m \in A \backslash B \\ m = e_m & m \in B \end{array} \right\}\end{array}$$

$$\begin{array}{c}\mathbf{new}\ c\ e \leadsto \mathbf{constr}\ Y(c)\ e \\ \mathbf{let}\ x\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2 \leadsto (\lambda x.e_2)\ e_1 \\ \lambda\langle x, y\rangle.e \leadsto \lambda z.\mathbf{let}\ \langle x, y\rangle\ \mathbf{be}\ z\ \mathbf{in}\ e\end{array}$$

Figure 7: Translation of Derived Forms

3 Operational Semantics

We define a big-step evaluation semantics using *heaps*, where an expression in some heap evaluates to a value and an updated heap. We restrict to expressions all of whose free variables are of object type, and identify these free variables l with *heap locations.*

In Figure 8 we give an evaluation relation $\Downarrow \subseteq (H \times E) \times (H \times V)$ from expressions with heaps to values with heaps, writing $h, e \Downarrow h', v$. We interpret a heap as an element of $H = L \rightharpoonup V \times V$, a partial function mapping locations to heap cells, where a heap cell is the actual object state paired with its class definition. So for $l \in \mathrm{dom}(h)$, $h(l) = (s, c)$ with s and c values representing the state and class of the object l. It is only the state which we will update, and so must essentially live in the heap, but it is convenient to store the class expression there, and one could imagine modelling method update in this way (although our denotational semantics could not interpret this).

Note that in the current presentation, the state consists of an object-type and ground-type part, only one of which is directly updated. Correspondingly, the method invocation rule updates the ground-type component of the state only.

We use the state convention that a rule not mentioning heaps

$$\frac{e_1 \Downarrow v_1 \cdots e_n \Downarrow v_n}{e \Downarrow v}$$

stands for the rule

$$\frac{h_0, e_1 \Downarrow h_1, v_1 \cdots h_{n-1}, e_n \Downarrow h_n, v_n}{h_0, e \Downarrow h_n, v}$$

as only object construction and method invocation need to interact with the heap.

Note that our operational semantics is untyped—types are not required at run time for our language. Also, here we can interpret a larger language than our games model, including cyclic heaps.

4 Games

Here we shall use the simplest notion of games we can get away with. As defined by Lamarche [Lam92] (and described in [Mel]), a game is simply a set of moves partitioned into opponent and player moves, together with the set of valid plays of that game. In particular there is no notion of questions and answers, or justification pointers, and these are not required for the purpose of our definitions. This leads to a linear category of games, on which we define an exponential, and which we then enlarge to give a setting for call-by-value computation.

We briefly review the definition and some of the structure of this category; full details will appear in [Wol].

A game A is given as $\langle M_A, \lambda_A, P_A \rangle$: a countable set of moves M_A, a labelling function $\lambda_A \colon M_A \to \{O, P\}$ (call the negation $\overline{\lambda}_A$), and a non-empty prefix-closed set of positions $P_A \subseteq M_A^*$, where each $s \in P_A$ consists of alternating O and P moves, starting with an O move. A strategy for A consists of a non-empty

$$\frac{}{v \Downarrow v} \qquad \frac{e \Downarrow \langle n_1, \ldots, n_k \rangle}{c_\varphi\ e \Downarrow m}\ \varphi(n_1, \ldots, n_k) = m$$

$$\frac{e \Downarrow 0 \quad e_1 \Downarrow v}{\textbf{ifz}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow v} \qquad \frac{e \Downarrow n \quad e_2 \Downarrow v}{\textbf{ifz}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow v}\ n \neq 0$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\langle e_1, e_2 \rangle \Downarrow \langle v_1, v_2 \rangle} \quad \frac{e \Downarrow \langle v_1, v_2 \rangle \quad e'[v_1/x, v_2/y] \Downarrow v}{\textbf{let}\ \langle x, y \rangle\ \textbf{be}\ e\ \textbf{in}\ e' \Downarrow v}$$

$$\frac{e_1 \Downarrow \lambda x.e' \quad e_2 \Downarrow v' \quad e'[v'/x] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{e \Downarrow \textbf{obj}\ \{m_1 = v_1, \ldots, m_n = v_n\}}{e.m_i \Downarrow v_i}\ 1 \leq i \leq n$$

$$\frac{e \Downarrow v}{Y(e) \Downarrow Y(v)} \quad \frac{e \Downarrow Y(v)}{e.m \Downarrow Y(v).m}$$

$$\frac{e_1 \Downarrow Y(v').m \quad (v'\ Y(v')).m\ e_2 \Downarrow v}{e_1\ e_2 \Downarrow v} \quad \frac{e_1 \Downarrow Y(v') \quad (v'\ Y(v'))\ e_2 \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{h, e_c \Downarrow h', v_c \quad h', e_v \Downarrow h'', v_v}{h, \textbf{constr}\ e_c\ e_v \Downarrow h''[l \mapsto \langle v_c, v_v \rangle], l}\ l\ \textit{fresh} \qquad \frac{e \Downarrow l}{e.m \Downarrow l.m}$$

$$\frac{h, e_1 \Downarrow h', l.m \quad h', v_c.m\ \langle v_o, v_b, e_2 \rangle \Downarrow h'', \langle v_b', v \rangle}{h, e_1\ e_2 \Downarrow h''[l \mapsto \langle\langle v_o, v_b' \rangle, v_c \rangle], v}\ h(l) = \langle\langle v_o, v_b \rangle, v_c \rangle$$

Figure 8: Operational Semantics

even-prefix-closed $\sigma \subseteq P_A$, where $s \in \sigma$ is even-length and σ is deterministic, that is for any $sa, sb \in \sigma$, $a = b$.

We define constructions on games A, B:

$$\begin{array}{rcl} M_{A\otimes B} & = & A+B \\ \lambda_{A\otimes B} & = & [\lambda_A, \lambda_B] \\ P_{A\otimes B} & = & \{s \in L_{A\otimes B} \mid \\ & & s\upharpoonright_A \in P_A, s\upharpoonright_B \in P_B\} \end{array} \qquad \begin{array}{rcl} M_{A\multimap B} & = & A+B \\ \lambda_{A\multimap B} & = & [\overline{\lambda}_A, \lambda_B] \\ P_{A\multimap B} & = & \{s \in L_{A\multimap B} \mid \\ & & s\upharpoonright_A \in P_A, s\upharpoonright_B \in P_B\} \end{array}$$

where L_A is the set of valid (i.e. correctly alternating) moves for A.

Our base category $\mathscr{C}$ has as objects these games, and morphisms $A \to B$ all strategies for $A \multimap B$. As usual identity is defined as a copycat strategy, and composition as "parallel composition plus hiding". One can easily show $\mathscr{C}$ with $\otimes$ to be an SMCC; additionally, $\mathscr{C}$ is affine, has an additive product & and a *skewed product* $A \oslash B$, defined as $A \otimes B$ but where the first move must be in A.[3]

Before proceeding, we note that products in our category behave in an *interfering* fashion, that is play in one component can affect future behaviour of the other. In other words, a strategy for $A \otimes B$ need not be a pair of strategies for A and B, and it is this which enables us to model stateful behaviour in $\mathscr{C}$.

We can extend these product operators to labelled and infinitary versions, and indeed we shall use an infinitary version of $\oslash$ as a *linear exponential*, intuitively defined as

$$!A = \mu X.\ A \oslash X$$

although we shall do without defining a μ operation for now. Concretely, take

$$\begin{array}{rcl} M_{!A} & = & \bigsqcup_{i\in N} M_A^{(i)} \\ \lambda_{!A} & = & \bigsqcup_{i\in N} \lambda_A \\ P_{!A} & = & \{t \in L_{!A} \mid \forall i.t\upharpoonright_{M_A^{(i)}} \in P_A\} \end{array}$$

We then define the required operations

$$\begin{array}{rclrcl} \delta_A & : & !A \to !!A & d_A & : & !A \to !A \otimes !A \\ \varepsilon_A & : & !A \to A & m_A & : & !A \otimes !B \to !(A \otimes B) \end{array}$$

These morphisms (excluding ε) are interesting in that they are defined in a *dynamic* copycat fashion—in the case of d, for example, the first move in a given !-component in the first or second $\otimes$-component of the right hand side sets up a connection with the first unplayed !-component on the left, after which d behaves in a copycat fashion with respect to those !-components. An important feature of our approach is that much of the complexity of the control flow of programs is isolated with these morphisms and is handled automatically by them.

We can use the CPO-structure of strategies with inclusion to define a fixed point operator on $\mathscr{C}$. The external fixed point operator is standard, but it is worth noting the type of the internal version

$$Y_A : !(A \multimap A) \to A$$

[3]This is the same as the $\oslash$ of [Lai02], but in reverse, i.e. by $A \oslash B$ we mean "A then B" instead of "A after B". Additionally, the exponential in [Lai02] is similarly constructed from the $\oslash$, although it should be noted that the different notion of games means the various forms of product there are all more "relaxed" than ours.

An exponential appears because the function has to be used repeatedly to obtain the fixed point.

We construct a category $\mathscr{D}$ to model values by a simplification of the $\mathbf{Fam}(\mathscr{C})$ construction of [AM98], essentially considering just the subcategory of $\mathbf{Fam}(\mathscr{C})$ where games are "families" of one repeated object $\{A \mid i \in I\}$. Objects of $\mathscr{D}$ are pairs (A, X) of a countable set A and an object X of $\mathscr{C}$, and morphisms $\langle \bar{f}, \hat{f} \rangle \colon (A, X) \to (B, Y)$ are pairs of functions $\bar{f} \colon A \to B$ and $\hat{f} \colon A \to \mathscr{C}(X, Y)$, and

$$\begin{gathered} \langle \bar{g}, \hat{g} \rangle \circ \langle \bar{f}, \hat{f} \rangle = \langle \bar{g} \circ \bar{f}, \hat{h}, \rangle \\ \hat{h}(n) = \hat{g}(\bar{f}(n)) \circ \hat{f}(n) \end{gathered}$$

We define $\multimap \colon \mathscr{D} \times \mathscr{C} \to \mathscr{C}$ as

$$(A, X) \multimap Y = X \multimap \&_A Y$$

To define lifting we first define in $\mathscr{C}$ a weak co-product $\Sigma_A X$ where there is an initial question followed by a response from A, followed by play in X, i.e.

$$P_{\Sigma_A X} = \{\varepsilon, q\} \cup \{qas \mid a \in A, s \in X\}$$

and then the lifting functor $\bot \colon \mathscr{D} \to \mathscr{C}$ simply takes $\bot(A, X) = \Sigma_A X$. The corresponding definition on morphisms, and structure to make $\bot$ a strong monad

$$\eta_A \colon A \to A_\bot \quad \mu_A \colon A_{\bot\bot} \to A_\bot \quad \Psi_{A,B} \colon A_\bot \otimes B_\bot \to (A \otimes B)_\bot$$

are easily defined. We shall also use the following notation for other available structure:

$$\mathit{cong} : !A \otimes !B \cong !(A \& B)$$

$$\frac{f \colon !A \to B_\bot}{f^\ddagger \colon !A \to (!B)_\bot} \qquad \frac{g \colon A \to B_\bot}{g^\dagger \colon A_\bot \to B_\bot}$$

To model natural numbers, we take $N = (\mathbb{N}, 1_{\mathscr{C}})$, and then for a function $\varphi \colon \mathbb{N}^k \rightharpoonup \mathbb{N}$ take $\bar{\varphi} \colon \otimes_{1,\ldots,k} N \to N_\bot$ the obvious morphism. Define conditionals

$$\mathit{ifz}_A \colon N_\bot \otimes (A_\bot \& A_\bot) \to A_\bot$$

One can also identify a "well-bracketed" submodel $\mathscr{C}_{wb}$ of $\mathscr{C}$ (and similarly $\mathscr{D}_{wb}$) by the use of additional structure, not described here. Moreover, all the pieces of structure mentioned above have analogues in $\mathscr{C}_{wb}$ and $\mathscr{D}_{wb}$, so that the definition of $[\![-]\!]$ given below can be understood as providing a semantics for our language in either $\mathscr{D}$ or $\mathscr{D}_{wb}$.

5 Denotational Semantics

We give a semantics in $\mathscr{D}$ of the form

$$[\![\Gamma \vdash e \colon \tau]\!] \colon [\![\Gamma]\!] \to [\![\tau]\!]_\bot$$

Definitions of $[\![-]\!]$ for types and terms are given in Figures 9–10; we additionally define the interpretation of the subtyping relation $[\![\tau <: \tau']\!] : [\![\tau]\!] \to [\![\tau']\!]$ via the appropriate projection morphisms $\Pi_{[\![\tau]\!],[\![\tau']\!]}$.

$$\begin{array}{rcl}
\llbracket N \rrbracket & = & !N = N \\
\llbracket \tau \to \tau' \rrbracket & = & \llbracket \tau \rrbracket \multimap \llbracket \tau' \rrbracket_{\perp} \\
\llbracket \sigma \otimes \tau \rrbracket & = & \llbracket \sigma \rrbracket \otimes \llbracket \tau \rrbracket \\
\llbracket \mathbf{Obj}\ \{m_1 = \tau_1, \ldots, m_n = \tau_n\} \rrbracket & = & !\&_{m \in \{1 \ldots n\}} \llbracket \tau_m \rrbracket \ \cong\ \otimes_{m \in \{1 \ldots n\}} ! \llbracket \tau_m \rrbracket
\end{array}$$

Figure 9: Denotation of Types

$$\begin{array}{rcl}
\llbracket x\colon \tau \vdash x\colon \tau \rrbracket & = & \eta_{\llbracket \tau \rrbracket} \\
\llbracket \Gamma \vdash e\colon \tau \rrbracket & = & \llbracket \Gamma, \Delta \vdash e\colon \tau \rrbracket \circ (id_{\llbracket \Gamma \rrbracket} \otimes !_{\llbracket \Delta \rrbracket}) \\
\llbracket \Gamma, y\colon \tau_2, x\colon \tau_1, \Delta \vdash e\colon \tau \rrbracket & = & \llbracket \Gamma, x\colon \tau_1, y\colon \tau_2, \Delta \vdash e\colon \tau \rrbracket \circ \\
 & & \quad (id_{\llbracket \Gamma \rrbracket} \otimes twist_{\llbracket \tau_1 \rrbracket, \llbracket \tau_2 \rrbracket} \otimes id_{\llbracket \Delta \rrbracket}) \\
\llbracket \vdash c_{\varphi}\colon N \otimes \ldots \otimes N \to N \rrbracket & = & \eta \circ \lambda(\bar{\varphi}) \\
\llbracket \Gamma, \Delta \vdash \mathbf{ifz}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\colon A \rrbracket & = & ifz_A \circ (\llbracket \Gamma \vdash e\colon N \rrbracket \otimes \\
 & & \quad \llbracket \Delta \vdash e_1 : A \rrbracket \& \llbracket \Delta \vdash e_2 : A \rrbracket) \\
\llbracket \Gamma, \Delta \vdash \langle e, e' \rangle\colon \tau \otimes \tau' \rrbracket & = & \Psi \circ (\llbracket \Gamma \vdash e\colon \tau \rrbracket \otimes \llbracket \Delta \vdash e'\colon \tau' \rrbracket) \\
\llbracket \Gamma, \Delta \vdash \mathbf{let}\ \langle x, y \rangle\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2\colon \tau \rrbracket & = & \llbracket \Gamma, x\colon \tau_1, y\colon \tau_2 \vdash e_2 : \tau \rrbracket^{\dagger} \circ \Psi \circ \\
 & & \quad (\eta_{\llbracket \Gamma \rrbracket} \otimes \llbracket \Delta \vdash e_1 : \tau_1 \otimes \tau_2 \rrbracket) \\
\llbracket \Gamma \vdash \mathbf{obj}\ \{m_1 = v_1, \ldots, m_n = v_n\}\colon \ \ldots \rrbracket & = & \perp(cong^n) \circ \Psi^n \circ \\
 & & \quad (\bigotimes_{1 \leq i \leq n} \llbracket \Gamma \vdash v_i : \tau_i \rrbracket^{\ddagger} \circ d^n) \\
\llbracket \Gamma, z\colon X \vdash e[z/x, z/y]\colon \tau' \rrbracket & = & \llbracket \Gamma, x\colon X, y\colon X \vdash e\colon \tau' \rrbracket \circ \\
 & & \quad (id_{\llbracket \Gamma \rrbracket} \otimes d_{\llbracket X \rrbracket}) \\
\llbracket \Gamma \vdash e.m\colon \tau \rrbracket & = & \perp(\Pi_m \circ \varepsilon) \circ \llbracket \Gamma \vdash e\colon \mathbf{Obj}\ \{m\colon \tau\} \rrbracket \\
\llbracket \Gamma \vdash \lambda x.e\colon \tau \to \tau' \rrbracket & = & \eta \circ \lambda_{\tau}(\llbracket \Gamma, x\colon \tau \vdash e\colon \tau' \rrbracket) \\
\llbracket \Gamma, \Delta \vdash e_1\ e_2\colon \tau \rrbracket & = & eval^{\dagger} \circ \Psi \circ \\
 & & \quad \llbracket \Gamma \vdash e_1 : \sigma \to \tau \rrbracket \otimes \llbracket \Delta \vdash e_2 : \sigma \rrbracket \\
\llbracket \Gamma \vdash Y(e)\colon X \rrbracket & = & Y \circ \llbracket \Gamma \vdash e\colon X \to X \rrbracket^{\ddagger} \\
\llbracket \Gamma, \Delta \vdash \mathbf{constr}\ e_1\ e_2\colon \tau \rrbracket & = & thread^{\dagger} \circ \Psi \circ \\
 & & \quad (\llbracket \Gamma \vdash e_1 : \tau_1 \rrbracket \otimes \llbracket \Delta \vdash e_2 : \tau_2 \rrbracket) \\
\llbracket \Gamma \vdash e\colon \tau' \rrbracket & = & \llbracket \tau <: \tau' \rrbracket \circ \llbracket \Gamma \vdash e\colon \tau \rrbracket
\end{array}$$

Figure 10: Denotation of Terms

Note that unlike our operational semantics, we have no notion of a heap here, just giving the denotation of a term in context. Stateful behaviour of objects is instead modelled by the behaviour of strategies of ! type. The clause in our interpretation for contraction uses the diagonal morphism d: $!A \to !A \otimes !A$, and this correctly manages any possible interference arising from multiple uses of the same object.

The morphism *thread* appearing in the clause for **constr** lies at the heart of our interpretation and requires some explanation. For simplicity, consider the construction of an object with a single method m whose concrete implementation has type $\sigma \otimes \gamma \otimes \tau \to \gamma \otimes \tau'$, where σ is a reusable type representing the "object part" of the internal state, and γ represents the ground type part of the state. This method implementation receives a denotation of type

$$!S \otimes T \otimes X \multimap (T \otimes Y)_\perp$$

where $!S = [\![\sigma]\!]$, $T = [\![\gamma]\!]$, $X = [\![\tau]\!]$, and $Y = [\![\tau']\!]$. (The reasons for not including $!S$ on the right hand side here will be discussed in Section 7.) In addition, we have the initial state of the object which is passed in as the second argument to **constr**; this will receive a denotation of type $!S \otimes T$. From these two data, it is possible to determine the abstract (externally visible) behaviour of the constructed object as a strategy of type $!(X \multimap Y_\perp)$, which captures the object's behaviour under any number of invocations of the method m. It is this "abstraction of behaviour" that is performed by the morphism *thread*:

$$\mathit{thread}_{S,T} : (!S \otimes T) \otimes\ !(!S \otimes T \otimes X \multimap (T \otimes Y)_\perp) \to !(X \multimap Y_\perp)_\perp$$

Intuitively, in the case of successive invocations of m, *thread* applies the concrete method implementation as many times as necessary, starting from the initial state and threading the updated state through the computation in the appropriate way. However, there is also the possibility that several invocations of m may be active at once, even in a non-concurrent setting. For instance, suppose we pass as an argument to m a function which, when applied, itself invokes m on the object in question. For such nested invocations, some care is required: the value of the (ground-type) state supplied as input to a method call should always be the value from the last time the state was updated — that is, from the most recent time we *returned* from an invocation of m (recall the discussion in Section 2).

The part of *thread* concerned with $!S$ is easy to implement — it is simply a case of taking many copies of $!S$ and supplying one copy to each instance of the method implementation; the interference between the components of $!S$ then automatically takes care of the propagation of any changes to the objects. The ground type part of *thread*, on the other hand, is defined recursively in a highly dynamic fashion. Let us regard T simply as a set, and consider the task of defining

$$\mathit{thread}_T : T \otimes\ !(T \otimes X \multimap (T \otimes Y)_\perp) \to !(X \multimap Y_\perp)_\perp$$

for a given "current state" $t \in T$ and a given computation involving multiple invocations of m. We focus attention on the first invocation of m (call this the *main invocation*), and divide subsequent invocations into two categories: those nested within the main invocation, and those occurring afterwards. We recursively handle the first group of invocations by supplying the state t as the input to this subcomputation. When the main invocation completes, it updates the state to some t', and we then recursively handle the second group of invocations supplying the state t' as input.

More formally, we first define for each $t \in T$ and $i \in \mathbb{N}$ a morphism

$$\mathit{prethread}_{ti} : Z \oslash \left(\bigotimes_{u \in T, j \in \mathbb{N}} !Z \right) \to !Z$$

where $Z = X \multimap (T \otimes Y)_\perp$. The whole family of morphisms is obtained via a simultaneous least fixed point, taking $\mathit{prethread}_{ti}$ to be the composition

$$\begin{tikzcd}
Z \oslash \bigotimes_{u,j} !Z \arrow[d, "id \oslash ((\bigotimes_{u,j} \mathit{prethread}_{uj}) \circ \mathit{reshuffle})"] & & \\
Z \oslash \bigotimes_{u,j} !Z \arrow[r, "\mathit{branch}_t"'] & Z \oslash !Z \arrow[r, "\mathit{fold}"'] & !Z
\end{tikzcd}$$

where *reshuffle* performs some complicated redistribution of components, and branch_t supplies components of $!Z$ by drawing them from the component $(t, 0)$ of the tensor product until such time as the initial question in the left-hand copy of Z has been answered by some move t', after which they are drawn from the component $(t', 1)$. The required strategy for thread_T at a given $t \in T$ is then readily constructed from $\mathit{prethread}_{t0}$. Full details will appear in [Wol].

6 Adequacy

Having given a language with a denotational semantics and an operational semantics which the reader will hopefully find intuitive, we naturally wish to show that these agree, or in the usual terminology that our denotational semantics is *adequate*. We describe some interesting aspects of the proof here, since even the soundness part has not been straightforward.

Firstly, our operational semantics is defined as a relation on heaps and expressions, while our denotational semantics only considers terms in context, having no notion of a heap. We do not have to extend the denotational semantics to terms in heaps—instead, we consider locations in an expression to be variables in an appropriate context. The denotation of the heap cell $(l \mapsto c, v)$ is taken to be the object $[\![\mathbf{constr}\ c\ v]\!]$, and this is composed with the denotation of the expression to give the interpretation of the variable l.

Since the operational semantics is untyped, we give a typing on heaps, and its interpretation, in Figure 11. In the case of flat heaps, this reduces to

$$[\![h = l_1 \mapsto (c_1, v_1), \ldots, l_n \mapsto (c_n, v_n)]\!] =$$
$$[\![\mathbf{constr}\ c_1\ v_1]\!] \otimes \ldots \otimes [\![\mathbf{constr}\ c_n\ v_n]\!]$$

We then interpret terms in heaps, where $\Delta \vdash h, e \colon \tau$

$$[\![e]\!]([\![h]\!]) = [\![\Phi(\Delta) \vdash e \colon \tau]\!] \circ [\![\Delta \vdash h]\!] : 1 \to [\![\tau]\!]_\perp$$

$$\Phi(\ \sigma \otimes \textbf{Obj}\ \{m\colon \sigma\otimes\tau \to \sigma\otimes\tau'\}_{m\in X}\ ) = \textbf{Obj}\ \{m\colon \tau \multimap \tau'\}_{m\in X})$$

$$\frac{}{\emptyset \vdash \emptyset} \qquad \frac{\Delta \vdash h \quad \Phi(\Delta) \vdash v\colon \tau}{\Delta, l\colon \tau \vdash h, l \mapsto v}$$

$$\begin{array}{rcl} [\![\emptyset \vdash \emptyset]\!] & = & id_1 \\ [\![\Delta, l\colon \tau \vdash h, l \mapsto (c,s)]\!] & = & [\![\Delta \vdash h]\!]; d; (id_{[\![\Phi(\Delta)]\!]} \otimes [\![\Phi(\Delta) \vdash \textbf{constr}\ c\ s\colon \tau]\!]) \end{array}$$

$$\Delta \vdash h, e\colon \tau \Leftrightarrow \Delta \vdash h \wedge \Phi(\Delta) \vdash e\colon \tau$$

Figure 11: Typing and interpretation of heaps

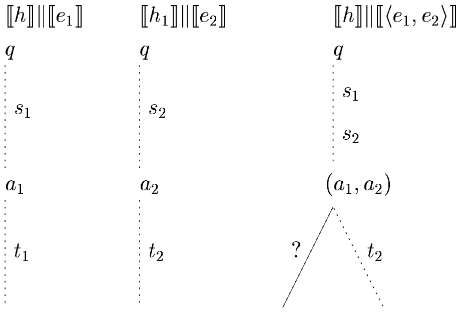

Figure 12: Pairing Interaction

Theorem 1 (Soundness) *If* $\Delta \vdash h, e \Downarrow \Delta' \vdash h', v$ *then* $[\![e]\!]([\![h]\!]) = [\![v]\!]([\![h']\!])$.

In order to prove soundness, we shall wish to compare strategies for values with strategies for expressions, and in particular after an expression performs some interaction with the heap we wish to be able to ask if the strategy *from that point onwards* is just the same as the strategy for the corresponding value given by the operational semantics. Note that it is not enough to ask that every play in $[\![e]\!]$ with its initial heap-interaction removed is a play in $[\![v]\!]$, for one needs to know what happens on both sides after some arbitrary interaction with the heap is the same. Consider evaluation of $h, \langle e_1, e_2\rangle$, depicted in Figure 12. If $h, e_1 \Downarrow h_1, v_1$ and $h_1, e_2 \Downarrow h_2, v_2$, we need to know that $[\![e_1]\!]$ after interacting with $[\![h_1]\!]$ would behave the same as $[\![v_1]\!]$ if the heap were to thereafter behave as $[\![h_2]\!]$.

For this we define a kind of memoization operation on strategies and heaps, as depicted in Figure 13, so that for an interaction sequence $s \in [\![h]\!] || [\![e]\!]$, $[\![e]\!]_s$ is as $[\![e]\!]$ but with heap-interaction s "cut out", and $[\![h]\!]^s$ is the heap that would result after s, so that

$$[\![e]\!]([\![h]\!]) = [\![e]\!]_s([\![h]\!]^s)$$

We think of $[\![e]\!]_s$ as the value resulting from evaluating e in heap h, and thus come to the following lemma to prove soundness. We use projections $\Pi_{[\![\Delta]\!]}$, $\Pi_{[\![\Delta']\!]}$ to handle growing heaps, the new part of the heap being supplied to the value $[\![v]\!]$ before comparing with $[\![e]\!]_s$.

In fact here we shall consider a simplification. In general even the types of $[\![e]\!]_s$ and $[\![v]\!]$ differ, as the former may interact further with the result of a

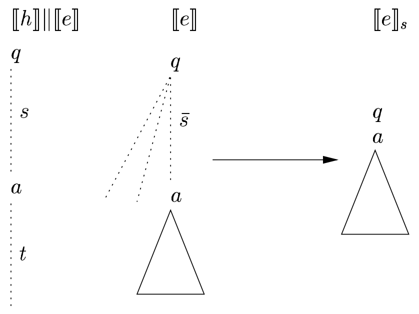

Figure 13: Memoization

method invocation in the interaction s, while this behaviour would be "hard-wired" in the latter. This can be accounted for, but for now we present the following definition for the case when this does not occur:

Lemma 2 (Soundness) *If*

$$\Delta \vdash h, e \Downarrow \Delta, \Delta' \vdash h', v$$

then there exists (unique) $qsa \in [\![h]\!] || [\![e]\!]$ *with*

$$\begin{array}{rcl} [\![h]\!]^s & = & \Pi_{[\![\Delta]\!]} \circ [\![h']\!] \\ [\![e]\!]_s & = & [\![v]\!](\Pi_{[\![\Delta']\!]} \circ [\![h']\!]) \end{array}$$

With this formulation, soundness follows via a straightforward induction on operational semantics derivations. However, a subtlety arises with respect to substitution. The problem is that the usual substitution lemma is not valid in a linear lambda calculus, as noted in [Wad91], in a semantics like ours where ! and !! are not isomorphic. Fortunately, in a call-by-value setting, one only requires substitution for values. We are able to show that *values* of object type are always denoted by a promoted morphism (unlike general expressions of that type), and this is all that is required for our substitution lemma to hold.

At present, the details of our soundness proof are only fully worked out for a somewhat restricted subset of the language described above (the most important restriction being to consider only objects with ground-type state, and hence flat heaps). However, we expect to be able to extend the proof to the language as a whole without difficulty. One would also like to show the other half of adequacy:

$$h, e \Uparrow \;\Rightarrow\; [\![e]\!]([\![h]\!]) = \bot$$

The proof is expected to involve a sophisticated use of logical relations.

We expect also to be able to show that the interpretation in $\mathscr{D}_{wb}$ is universal and fully abstract, that is:

$$\forall a : 1 \to [\![\tau]\!].\ \exists(\emptyset \vdash e : \tau).\ [\![\emptyset \vdash e : \tau]\!] = a$$

$$(\forall C[-], v.\, C[e] \Downarrow v \Leftrightarrow C[e'] \Downarrow v) \;\Rightarrow\; [\![\Gamma \vdash e : \tau]\!] = [\![\Gamma \vdash e' : \tau]\!]$$

We believe that both of these properties can be shown by the following route. Observe that for any type τ, well-bracketed strategies of type τ can be encoded by partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$, which in turn can be represented by programs of a certain object type ρ. One can then construct a program $\mathit{interpret}_\tau : \rho \to \tau$ with the property that

$$\forall e : \rho, a : 1 \to [\![\tau]\!].\ (e \text{ codes } a) \;\Rightarrow\; [\![\mathit{interpret}_\tau\ e]\!] = a$$

The construction of these programs proceeds by induction on the structure of τ. Both universality and full abstraction follow readily from the existence of such programs.

7 Extensions of our language

7.1 Adding new fields in subclasses

There is an important deficiency in the translation of classes we described earlier. Normally when one creates a subclass, one would expect to add new fields as well as methods, however we have not allowed for this. The problem is that the state appears as both argument to and result of the step function being extended.

Various extensions to our calculus can be used to tackle this problem. When defining a class, we must give the step function a type which includes not only the state of that class, but the *potential subclass state*. When the class is extended, that potential subclass state will be partially instantiated as the newly added state, together with the *new* potential additional state; when the class is instantiated, the additional state is taken to be something trivial.

By treating the potential future state polymorphically, we thus cater for any particular choice of additional state when a class is extended. This can of course be implemented by adding general polymorphism to our language. However, our category of games has a universal object, so we could choose to use this to include some facility in our language to allow for the interpretation suggested above.

7.2 An extension with control operators

As we have remarked, although the notion of well-bracketing is not required for the definition of our semantics, the denotations of all terms in our language turn out to live within the well-bracketed submodel $\mathscr{D}_{wb}$ of $\mathscr{D}$, and it is with respect to this submodel that full abstraction and universality properties must be formulated. It is therefore natural to ask whether there is some natural extension of the language corresponding to $\mathscr{D}$ itself.

We here briefly outline how such an extension can be obtained by adding a continuation-style operator to our language. For simplicity, let us pretend that our language has sum types (which could indeed be added if we made use of the general $\mathbf{Fam}(\mathscr{C})$ construction). We may then add an operator **catchcont** with the following typing rule:

$$\frac{\Gamma \vdash e\colon \mathbf{Obj}\ \{m\colon \tau_1 \to \tau_2\} \to \tau_3}{\Gamma \vdash \mathbf{catchcont}\ e\colon \tau_3 + (\tau_1 \otimes (\tau_2 \to \mathbf{Obj}\ \{m\colon \tau_1 \to \tau_2\} \to \tau_3))}$$

where τ_1, τ_2, τ_3 are all ground types. Intuitively, the evaluation of **catchcont** e proceeds by attempting to evaluate $e\ z$, where z is a dummy argument of type **Obj** $\{m\colon \tau_1 \multimap \tau_2\}$. If we can evaluate this to some value $v_3\colon \tau_3$ without ever having to know what z stands for, $\mathbf{inl}(v_3)$ is returned. However, if the computation is blocked because the value of some expression $z.m(v_1)\colon \tau_2$ is required in order to make further progress (where v_1 is a value of type τ_1), we suspend the computation flagging up the requested value v_1 together with an operation allowing us to resume the computation once a suitable value of type τ_2 is supplied. The linearity of the arrow type in question effectively means that the continuations we obtain are not reusable or copyable and can only be used in a linear fashion. As hinted at in the Introduction, this operator suffices to support various kinds of coroutining.

It is reasonably straightforward to extend our operational semantics to take account of **catchcont**, by introducing a class of *open values* that are allowed to contain variables together with a suitable notion of evaluation context. The denotational interpretation of **catchcont** is also fairly straightforward. Moreover, it can be shown that in the presence of **catchcont** we can syntactically define a retraction $(\upsilon \multimap \upsilon) \lhd \upsilon$, where υ is the universal type **Obj** $\{m : N \multimap N\}$. This is the key step required to show that every type is a definable retract of υ, and this provides a cheap route to full abstraction and universality properties along the lines explained in [Lonb]

The main gap that we have not yet filled concerns the proof of adequacy for the extended language. Though we fully expect adequacy to hold, the presence of **catchcont** greatly increases the complexity of control flow in the language, and it seems likely that it requires a significant strengthening of our (already quite elaborate) soundness proof. This may well throw up further unanticipated difficulties.

7.3 Adding pointer update

As we remarked in Section 2, we have so far been considering a kind of restricted method behaviour in which methods cannot directly update fields of object type by means of pointer assignment (though they may cause state changes to the objects they store by interacting with them). This means that once a portion of heap has been constructed, its topology remains fixed — fields of object type are stuck with the objects that were assigned to them when the enclosing object was constructed — and this may have created an impression that our approach is unable to deal with pointer updates or changing heap topologies of any kind.

In fact, certain special kinds of pointer updates can be incorporated without any fundamental change to our framework, though at the cost of a little more work. Specifically, it is fine to allow a method to update pointers provided the enclosing object already "possesses" the pointers in question or can generate them for itself. Thus, for instance, a method could swap the contents of two existing object fields, or could construct a brand new object to put in a field, but what it must not do is to "capture" a pointer that is passed in as (part of) the method argument. (The difficulty that arises if it does will be explained in the next subsection.)

Syntactically, this restriction on pointer manipulation can be enforced by allowing the method implementations in the rule for **constr** to have the more general type

$$\sigma \otimes \gamma \otimes \tau_m \to \sigma \otimes \gamma \otimes \tau'_m$$

but restricting the syntactic form of these method implementations to ensure that any dependence of the σ component of the result on the τ_m component of the input is mediated by some expression of ground type (which cannot conceal any pointers). This can be achieved in a somewhat *ad hoc* way by means of a complicated typing rule. On the denotational side, the construction of the morphism *thread* can be elaborated so that it takes appropriate care of the object type part of the state as well as the ground type part.

Because the details involved in this extension are somewhat messy, in the present paper we have chosen to content ourselves with a language without pointer update.

7.4 Pointer capture and beyond

Far more challenging is the question of how to model methods that do capture pointers in the above sense. The basic difficulty is that, even at the level of abstract behaviours, our choice of denotation for object types

$$[\![\mathbf{Obj}\ \{m_1 = \tau_1, \ldots, m_n = \tau_n\}]\!] \ = \ !\&_{m \in \{1 \ldots n\}}[\![\tau_m]\!] \ \cong \ \otimes_{m \in \{1 \ldots n\}}![\![\tau_m]\!]$$

is no longer adequate. The problem is illustrated by the following example. Suppose an object o possesses a method $m : \tau_1 \to \tau_2$, where τ_1 is an object type. Suppose moreover that an expression $o.m(o')$ has been evaluated—that we have returned from the call to m—and o has retained a pointer to o'. Since o' is now part of the state of o, any subsequent interactions with o might in principle trigger interactions with o'. However, such interactions will not be legal in terms of the rules of the above game: if we restrict our attention to the moves played in the copy of $[\![\tau_1 \to \tau_2]\!]$ corresponding to the first call to m, we would have an out-of-turn player move in $[\![\tau_1]\!]$ following on from a player move in $[\![\tau_2]\!]$.

One might respond to this problem in various ways. One solution would be to move to a different category of games, as in [AHM98] or [Lai02], where a more relaxed definition of pairs and exponentials allows the kind of behaviour illustrated above. Another solution would be to incorporate some notion of names (or pointers) into the semantics, and to make use of the indirection that these provide. Whilst both of these are interesting approaches, in our view the simplicity of our category, and the purity of our "behavioural" view of objects, have much to commend themselves, and it is interesting to ask whether pointer-capturing behaviour can be accommodated within this framework.

One approach which seems promising is to express the reusability of objects by means of a different choice of linear exponential. Recall that our operator ! can be considered to be defined by

$$!A \cong \mu X.A \oslash X$$

If we wish to capture the idea that with each new use of A we supply a value of some type B which remains available to all later uses of A, we might instead try something like

$$\mu X.B \multimap (A \oslash X)$$

or even better,

$$\mu X.!(B \multimap (A \oslash X))$$

since the latter allows us to take better account of nested activations of the object in question. However, this leads to significant technical difficulties, and we have not yet fully succeeded in getting this approach to work.

We are hopeful that using some approach of this kind, it may be possible to cater for all possible pointer manipulations involving only *acyclic* heaps. As indicated in the Introduction, to go beyond this it seems that we are forced to move to a semantic framework that takes some account of names.

References

[AC96] Martín Abadi and Luca Cardelli. *A theory of objects.* Springer, 1996.

[AGM+04] Samson Abramsky, Dan Ghica, Andrzej Murawski, Luke Ong, and Ian Stark. Nominal games and full abstraction for the nu-calculus. In *Proceedings of the Nineteenth Annual IEEE Symposium on Logic in Computer Science*, pages 150–159. IEEE Computer Society Press, 2004.

[AHM98] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references, 1998.

[AJM94] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. In *Theoretical Aspects of Computer Software*, pages 1–15, 1994.

[AM98] Samson Abramsky and Guy McCusker. Call-by-value games. In *CSL '97: Selected Papers from the 11th International Workshop on Computer Science Logic*, pages 1–17, London, UK, 1998. Springer-Verlag.

[BPSM99] Viviana Bono, Amit J. Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and objects. In *Fifteenth Conference on the Mathematical Foundations of Programming Semantics*, 1999.

[Lai02] James Laird. A categorical semantics of higher order store. *Electr. Notes Theor. Comput. Sci.*, 69, 2002.

[Lam92] F. Lamarche. Sequentiality, games and linear logic. In *Workshop on Categorical Logic in Computer Science.* Aarhus University, 1992.

[Lona] John Longley. A programming language based on game semantics. Grant proposal.

[Lonb] John Longley. Universal types and what they are good for. *Domain theory, logic and computation.*

[Mel] Paul-André Melliès. Sequential algorithms and strongly stable functions. *Theoretical Computer Science.* To appear.

[Wad91] P. Wadler. There's no substitute for linear logic. Manuscript, December 1991.

[Wol] Nicholas Wolverson. *Game semantics for object-oriented languages.* PhD thesis, University of Edinburgh.