WOOD 2004 Preliminary Version

Difunctorial Semantics of Object Calculus

Johan Glimming [1]

Department of Numerical Analysis and Computer Science
Stockholm University
Sweden

Neil Ghani [2]

Department of Mathematics and Computer Science
University of Leicester
United Kingdom

Abstract

In this paper we give a denotational model for Abadi and Cardelli's first order object calculus $\mathbf{FOb}_{1+\times\mu}$ (without subtyping) in the category pCpo. The key novelty of our model is its extensive use of recursively defined types, supporting self-application, to model objects. At a technical level, this entails using some sophisticated techniques such as Freyd's *algebraic compactness* to guarantee the exstence of the denotations of the object types.

The last sections of the paper demonstrates that the canonical recursion operator inherent in our semantics is potentially useful in object-oriented programming. This is witnessed by giving a straightforward translation of algebraic datatypes into so called wrapper classes. We conclude by comparing with Abadi and Cardelli's per semantics and by discussing our future work.

Key words: Category theory, object-oriented programming

1 Introduction

The semantics of objects is inherently complicated. Firstly, objects are recursive in the sense that they contain methods which operate on the object itself and may return the object as a result. This reference is known as the

[1] Email: `glimming@kth.se`
[2] Email: `ng13@mcs.le.ac.uk`

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

self parameter to the method. Secondly, object types are usually combined with a notion of subtyping, which can introduce anomalies in the operational semantics. Thirdly, method update (including inheritance) is difficult to model, particularly in combination with subtyping and binary methods. Fourthly, objects often come with some notion of class, which leads to the problem of finding an encoding of classes (pre-methods [1], *new* functions [23], etc have been studied) and an associated mechanism for creating new object instances from classes. Arguably, all these listed problems arise from the recursive nature of objects (see e.g. [3]).

There has been much research on finding good approaches to dealing with the recursion inherent in objects. *Self-application semantics [16,10]* takes the point of view that objects should be modelled as complicated recursive types. There are other approaches involving *higher-order polymorphism [17,18]* which hide all recursion under an existential quantifier. *Recursive record semantics [5,6,23]* can be seen as a compromise where (covariant) recursive types are used for self-returning methods, and a fix-point operator at the level of terms is needed to handle references to the object's instance variables. Unfortunately, a direct self-application encoding into $F^{\omega}_{<:}$ fails to support subtyping, while the other two listed approaches do not support method update (note, however, that [4] gives an encoding with both recursion and bounded existentials that does support method update). Abadi and Cardelli [1] proposed a variety of different object calculi which support method update and gave them a *primitive semantics* based upon reduction rules.

Our starting point is that method update is indeed an important ingredient in the object-oriented paradigm, and we believe that recursive types should be used in modelling objects. For reasons explained in [1] we therefore consider the notion of object fundamental, and hence prefer a streamlined theoretical approach rather than an encoding via $F^{\omega}_{<:}$. One aim of our work is indeed to provide a mathematical foundation for logics of object calculi, particularly logics of program transformations. Direct denotational models of object calculi are more suited to this purpose.

This paper presents the first steps in this program. We develop a categorical model for object calculus $\mathbf{FOb}_{1+\times\mu}$ presented in [1] (with minor modifications) based on an interpretation of object types as recursive types via mixed variance functors. The encoding of object types has the flavor of a self-application semantics. The main novelty with this model is in its extensive use of recursively defined types to model object types. Our mathematical setting is the category pCpo where we model objects as solutions of $self \cong self \to \mathsf{F}\, self$, where F is a covariant functor representing the type of the methods. The simplest objects give rise to a constant functor F while the full generality allows us to define what we term *wrapper classes* for algebraic datatypes. We believe that this model can be extended to support subtyping by natural transformations on the underlying functor F and, by regarding F as a *pattern functor*, opens the way to a polytypic style of object oriented programming.

The paper is structured as follows: after setting up the mathematical framework of Freyd's algebraically compact categories in section 2, we present the calculus $\mathbf{FOb}_{1+\times\mu}$ in section 3. In section 4 we give a semantics for the calculus in pCpo. In section 5, we discuss wrapper classes, i.e. link our semantics of objects to algebraic and coalgebraic style of programming, including giving the example of natural number objects. Finally, section 6 summarises our contribution and compares to related work. Finally, as this work is still in its early stages we are most happy to receive comments and feedback on directions and extensions.

2 Mathematical Preliminaries

We assume the reader is familiar with elementary category theory and in particular the basic concepts of category theory such as product, and exponential — see the bible [13] for details. As mentioned in the introduction, we propose a denotational model of typed object calculi whose key novelty is the use of recursively defined types. A simple example, e.g. finding an object D such that $D \cong [D, D]$, indicates that the existence of such recursively defined types is not at all obvious, e.g. there is clearly no set D such that $D \cong [D, D]$. The key feature of this example is that the mapping of an object D to the object $[D, D]$ is not a functor in that the left occurrence of D in the expression $[D, D]$ occurs *contravariantly* while the right occurrence is *covariant*. Such mappings are called *difunctors*.

Definition 2.1 [Difunctor] If $\mathcal{C}$ is a category, a difunctor is a functor $\mathsf{F} : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$. A fixed point of such a difunctor is an object X such that $X \cong \mathsf{F}\,X\,X$

There has been much research on finding fixed points for difunctors. The classic paper [21] defines a category of embedding and projection pairs where the functor F acts covariantly and from which a fixed point of F can be derived. More recently, [12,11,8,9] have used the more axiomatic setting of *algebraically compact* categories. i.e. categories where all (in a suitably qualified sense) covariant functors have an initial algebra the inverse of whose structure map is the final coalgebra. The related, but weaker, property of *algebraic completeness* merely requires all (again, in a suitably qualified sense) covariant functors to have an initial algebra.

The axiomatic approach is potentially easier to apply to non-domain theoretic models such as realizability models and models containing intensional features. Since we do not wish to over commit ourselves to a specific semantic setting at this stage, we therefore implicitly follow the axiomatic setting of [8,9] in working in the Kleisli category of a lifting monad. However, given the nature of the audience, we camouflage this general agenda by explicitly working in the canonical model of the category pCpo of w-complete partial orders and partial continuous functions. We denote by Cpo the subcategory of pCpo consisting of all cpos and total continuous functions. The salient facts about the categories pCpo and Cpo can be found in [19]. Cpo has the standard structure of being cartesian closed with finite coproducts. We give a brief summary of the structure of pCpo:

- **Zero object:** The empty cpo is a zero object in pCpo. That is, it is both an initial object and a terminal object.
- **Coproducts:** If A and B are cpos, their disjoint union is the coproduct of A and B in pCpo.
- **Partial Products:** If A and B are cpos, the cartesian product of the underlying sets is their partial product. It is not a product as the domain of definition of the pairing (f,g) is the intersection of the domains of f and g and hence $fst(f,g) \neq f$ etc. We denote the partial product by $A \otimes B$ to remind ourselves it is not a product.
- **Kleisli/Partial Exponentials:** If A and B are cpos, then the set of partial continuous functions from A to B forms a cpo as usual. We denote this cpo $[A,B]$ or $A \rightharpoonup B$. As expected, partial exponentials are right adjoint to the partial product. $-\otimes A \dashv [A,-] : \mathsf{Cpo} \longrightarrow \mathsf{pCpo}$. Note the domains and codomains for the functors involved in this adjunction.
- **Compactness:** pCpo is algebraically compact in that all locally continuous functors have coinciding initial algebras and final coalgebras [11].

It is worth making the observation here that, apart from compactness, we would have liked our ambient category to be cartesian closed and have finite coproducts so that we could manipulate polynomial functors and their (co-)algebras using the standard techniques. Indeed, settling for partial products and Kleisli exponentials may seem like a poor alternative. However, any compact category has a zero object (induced as the fixed point of the identity functor) and a CCC with a zero object is inconsistent as

$$A \cong A \times 1 \cong A \times 0 \cong 0$$

Hence we cannot get away from working in a non-cartesian closed setting. Nevertheless, the subcategory Cpo (where values take their denotation) is, of course, still cartesian closed.

So given a category like pCpo, how does one find fixed points for difunctors? Recall that in the simpler case of a covariant endofunctor $\mathsf{F} : C \longrightarrow C$, one finds an object $A \cong FA$ as the initial F-algebra or final F-coalgebra.

Definition 2.2 [Algebra, Coalgebra] Given a functor F we say that an arrow $\alpha : \mathsf{F}\,A \to A$ is an F-algebra with carrier A. Such F-algebras are the objects in a category **Alg**(F) for every functor F. The dual notion is that of F-coalgebra, i.e. reversed arrows $\alpha : A \to \mathsf{F}\,A$. The arrows between (co)algebras are F-homomorphisms, i.e. arrows h such that, for F-algebras the left diagram below commutes and, for F-coalgebras the right diagram below commutes:

$$\begin{tikzcd}
\mathsf{F}\,A \arrow[r, "F\,h"] \arrow[d, "\alpha"'] \arrow[dr, phantom, "Alg(\mathsf{F})"] & \mathsf{F}\,B \arrow[d, "\beta"] \\
A \arrow[r, "h"'] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "h"] \arrow[d, "\alpha"'] \arrow[dr, phantom, "Coalg(\mathsf{F})"] & B \arrow[d, "\beta"] \\
\mathsf{F}\,A \arrow[r, "F\,h"'] & \mathsf{F}\,B
\end{tikzcd}$$

When working with difunctors, algebras and coalgebras generalise to *dialgebras*. Note the presence of both covariance and contravariance in a difunctor means that we have no need for the dual notion of a dialgebra. The term dialgebra has several definitions in the literature, and so we give ours here:

Definition 2.3 [Dialgebras] A G-*dialgebra* for difunctor $\mathsf{G} : C^{op} \times C \to C$ is a pair of objects A, B together with an associated pair of arrows $f : \mathsf{G}\,A\,B \to B$ and $g : A \to \mathsf{G}\,B\,A$.

The category of dialgebras has maps between dialgebras given as follows

Definition 2.4 [Dialgebra Maps] Given G-dialgebras (A, B, ϕ, ψ) and (A', B', ϕ', ψ'), a G-*homomorphism* is a pair of arrows $(g : B \to B', h : A' \to A)$ such that the following diagrams commute:

$$\begin{tikzcd}
\mathsf{G}\,A\,B \arrow[r, "\phi"] \arrow[d, "\mathsf{G}\,g\,h"'] & B \arrow[d, "h"] \\
\mathsf{G}\,A'\,B' \arrow[r, "\phi'"'] & B'
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "\psi"] & \mathsf{G}\,B\,A \\
A' \arrow[r, "\psi'"'] \arrow[u, "g"] & \mathsf{G}\,B'\,A' \arrow[u, "\mathsf{G}\,h\,g"']
\end{tikzcd}$$

A key idea in axiomatic domain theory is to use algebraic compactness to find fixed points for difunctors. Here is a sketch of the construction:

Lemma 2.5 *Let* $\mathsf{G} : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$ *be a difunctor on an algebraically compact category* $\mathcal{C}$. *Then F has a fixed point.*

Proof. Form the functor $\mathsf{G}' : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}^{op} \times \mathcal{C}$ by following the *doubling trick* proposed by Freyd:

$$\mathsf{G}'\, X\, Y \triangleq (\mathsf{G}(Y, X), \mathsf{G}(X, Y))$$

Since $\mathcal{C}$ is algebraically complete, so is $\mathcal{C}^{op} \times \mathcal{C}$ and thus G' has an initial algebra, say $\mathsf{G}'(X, Y) \to (X, Y)$, which is given by maps $inn_G : X \to \mathsf{G}(Y, X)$ and $out_G : G(X, Y) \to Y$. By Lambek's lemma, inn_G and out_G are isomorphisms. Next, the pair $(out_G, inn_G) : (Y, X) \to \mathsf{G}'(Y, X)$ is easily seen to be the final G'-coalgebra. Since $\mathcal{C}$ is algebraically compact, so is $\mathcal{C}^{op} \times \mathcal{C}$ and hence the initial G'-algebra and final G'-coalgebra coincide. Thus $X = Y$ and we have a G fixed point as required. □

Of course, while the above proof may seem simple, much of the work is hidden in proving that i) algebraic completeness and compactness are preserved by taking products and opposite categories; ii) formalising exactly the class of difunctors which are to be considered; and iii) proving that certain categories are algebraically complete and compact. Further subtle and technical issues arise, e.g. that these fixed points should be suitably parameterised etc, but for this presentation we have decided to gloss over the details. See [8] for details. Having said this, the modularisation of the construction of fixed points is very elegant. Notice also that more is true than we claimed. In particular we constructed a specific fixed point of a difunctor with a universal property, namely, the initial dialgebra. We shall put this universal property to use later.

3 Object Calculus

We will now give the syntax and operational semantics of a first-order object calculus, henceforth referred to as **FOb**, which is essentially Abadi and Cardelli's $\mathbf{FOb}_{1\mu}$ extended with unit, product and coproducts. **FOb** hence has method updates, but not subtyping or higher-types. There are no real surprises in the calculus and its inclusion is merely for the sake of the completeness of the paper.

We assume countable sets L (method labels), V (type variables), and U (term variables) and will use Greek letters for type variables, lower case letters for term variables, and l_i with index $i \in \mathbb{N}$ for method labels.

Definition 3.1 [**FOb**-types] The set $\mathcal{T}_{\mathbf{FOb}}$ is defined by induction with

$$
\begin{array}{lll}
\tau ::= v & & \text{type variables} \\
\quad \mid & 1 & \text{terminal type} \\
\quad \mid & \tau_1 \times \tau_2 & \text{product types} \\
\quad \mid & \tau_1 + \tau_2 & \text{sum types} \\
\quad \mid & \tau_1 {\to} \tau_2 & \text{function types} \\
\quad \mid & \mu v.\tau & \text{recursive types} \\
\quad \mid & [l_1 : \tau_1, \ldots, l_n : \tau_n] & \text{object types}
\end{array}
$$

where $v \in V$, and for each i, $l_i \in L$.

Notice that there is no restriction on the occurrences of type variables in recursive types which means this calculus is expressive in including a variety of sophisticated types such as $\mathsf{Lam} = \mu X.\mathrm{N} + (X{\to}X)$ where $\mathrm{N} = \mu X.1 + X$. As usual, one next defines the pre-terms of **FOb**.

Definition 3.2 [**FOb**-preterms] The set $\mathcal{N}_{\mathbf{FOb}}$ is defined by induction with $U \subseteq \mathcal{N}_{\mathbf{FOb}}$. Inductively, if $m, m', b \in \mathcal{N}_{\mathbf{FOb}}$, $\tau_i \in \mathcal{T}_{\mathbf{FOb}}$, $l_i \in L$ and $x_i \in U$ (with $i \in \mathrm{N}$) then $*$, $[l_1 = \varsigma(x_1 : \tau_1)b_1, ..., l_n = \varsigma(x_n : \tau_n)b_n]$, $m.l$, $m\,m'$, $\lambda(x : \tau)b$, $m.l \Leftarrow \varsigma(x : \tau)m'$, $inl\,m, inr\,m$, $case(b, x.m, y.m')$, $fst\,m, snd\,m$, (m, m'), $inn(\tau, m)$, and $out(m)$ are in $\mathcal{N}_{\mathbf{FOb}}$.

We will adopt the following convention for meta-variables denoting terms: the symbol o is a term of the form $[l_1 = \varsigma(x_1 : \tau_1)b_1, ..., l_n = \varsigma(x_n : \tau_n)b_n]$ for some $n \in \mathrm{N}$. Sometimes we also write $[l_i = \varsigma(x_i : \tau_i)b_i]_{i \in I}$ for such terms, for I a finite subset of N. Terms of these two equivalent forms are known as *objects*.

For convenience we will identify any two terms (types) which are equal up to the order of method labels, e.g. $[l_1 = ..., l_2 = ...] \equiv [l_2 = ..., l_1 = ...]$, and we assume that method labels occurring inside the same enclosing brackets [...] are distinct. We use Abadi and Cardelli's definition of substitution, and write $m[a/b]$ meaning a is substituted for all free occurrences of b in m [1]. Further, $m(x)$ means x is free in m (hence possibly not even occurring in m), and then $m(a) \equiv m[a/x]$.

Definition 3.3 [**FOb**-Environments] An environment E is a finite sequence of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$ with no variable occurring twice in the sequence.

The typing judgments are of the form $E \vdash a : \sigma$ where E is an environment, a is a pre-term and σ is a type. We also let $E \vdash \sigma$ abbreviate $\exists a \in \mathcal{N}_{\mathbf{FOb}}$ such that $E \vdash a : \sigma$, i.e. the statement that σ is a well-formed inhabited type.

Definition 3.4 [**FOb**-Typing judgments] The typing judgments of **FOb** are

$$\frac{E, x_i : \sigma \vdash b_1 : \tau_1, ..., b_n : \tau_n \qquad \sigma = [l_1 : \tau_1, \ldots, l_n : \tau_n]}{E \vdash [l_1 = \varsigma(x_1 : \sigma)b_1, ..., l_n = \varsigma(x_n : \sigma)b_n] : \sigma} \text{(object intro)}$$

$$\frac{E \vdash a : \sigma, \qquad E, x : \sigma \vdash b : \tau_j \qquad \sigma = [l_1 : \tau_1, ..., l_n : \tau_n]}{E \vdash a.l_j \Leftarrow \varsigma(x : \sigma)b : \sigma} \text{(object update)}$$

$$\frac{E \vdash a : [l_1 : \tau_1, ..., l_n : \tau_n] \qquad 0 < i < n+1}{E \vdash a.l_i : \tau_i} \text{(object elim)}$$

$$\frac{E \vdash c : \sigma_1 + \sigma_2 \qquad E, x_i : \sigma_i \vdash m_i : \tau \qquad (i = 1, 2)}{E \vdash case(c, x_1.m_1, x_2.m_2) : \tau} \text{(case)}$$

$$\frac{E \vdash a : \tau}{E \vdash inl\, a : \tau + \sigma} \text{(inl)} \qquad \frac{E \vdash m : \tau_1 \to \tau_2, n : \tau_1}{E \vdash m\, n : \tau_2} (\lambda\text{-elim})$$

$$\frac{E \vdash b : \sigma}{E \vdash inr\, b : \tau + \sigma} \text{(inr)} \qquad \frac{E, x : \tau_1 \vdash b : \tau_2}{E \vdash \lambda(x : \tau_1)b : \tau_1 \to \tau_2} (\lambda\text{-intro})$$

$$\frac{E \vdash a : \tau \times \sigma}{E \vdash fst\, a : \tau} \text{(fst proj)} \qquad \frac{}{E \vdash * : \mathbb{1}} \text{(unit-unit)}$$

$$\frac{E \vdash a : \tau \times \sigma}{E \vdash snd\, a : \sigma} \text{(snd proj)} \qquad \frac{E \vdash b : \sigma[\mu\,\alpha.\sigma(\alpha)]}{E \vdash inn(\tau, b) : \mu\alpha.\sigma(\alpha)} (\mu\text{-in})$$

$$\frac{E \vdash a : \tau \qquad E \vdash b : \sigma}{E \vdash (a, b) : \tau \times \sigma} \text{(pair form)} \qquad \frac{E \vdash b : \mu\,\alpha.\sigma(\alpha)}{E \vdash out(b) : \sigma[\mu\,\alpha.\sigma(\alpha)]} (\mu\text{-out})$$

We say a pre-term $m \in \mathcal{N}_{\mathbf{FOb}}$ is well-typed iff there exists a type $\tau \in \mathcal{T}_{\mathbf{FOb}}$ and an environment E such that $E \vdash m : \tau$. We let $\mathcal{M}_{\mathbf{FOb}}$ denote the set of well-typed terms up to the obvious notion of α-equivalence induced by the term-binders λ, ς and *case*.

4 Difunctorial Semantics

In this section we give a denotational model of **FOb** using the category pCpo. The key feature of this semantics is that is reflects our intuition that the object types of **FOb** are fixed points of recursive type equations. More specifically, the recursion is over the self-parameter which occurs negatively. This intuition is clearly seen in the object-intro typing rule for $\sigma = [l_1 : \tau_1, \ldots, l_n : \tau_n]$ which suggests the i'th method will consume the self-parameter, which has type σ, to produce something of type τ_i. Thus, intuitively, the interpretation of σ should satisfy

$$[\![\sigma]\!] \cong [\![\sigma]\!] \rightharpoonup [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$$

and hence the denotation of σ should be the fixed point of $\mu X.X \rightharpoonup [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$. Crucially, the following lemma shows that such an interpretation supports *self-application* [10] which our semantics both requires and supports. We state the lemma specifically for pCpo to make clear we are not using cartesian closure in the proof.

Lemma 4.1 *Let* $\mathsf{F} : \mathsf{pCpo} \longrightarrow \mathsf{pCpo}$ *be a covariant functor and* O *satisfy* $O \cong [O, FO]$. *Then there is a self-application map sapp* $: O \to FO$.

Proof. All isomorphisms are total and hence the isomorphism uncurries to give a map $O \otimes O \to FO$. Now precompose with the diagonal which partial products posses. $\square$

Notice how this differs with the recursive record semantics [1], where the recursion is in the output or covariant position while the contravariant occurrence of *self* is replaced by having a separate state type, and a fixed point operator at the level of terms. Our semantics also differs from other encodings such as various encoding with existentials [18,4] where the contravariant occurrence is present but hidden under the existential quantifier. In our model of **FOb** we instead explicate the contravariant *self* parameter and interpret all object types into more elaborate recursive types which, as we have seen, support self-application.

If C is a category we denote by $\hat{\mathrm{C}}$ the category $C^{op} \times C$ and note that $(\hat{C})^n = (\hat{C^n})$. The doubling trick used to obtain fixed points of difunctors assigns to each difunctor $F : C^{op} \times C \longrightarrow \mathcal{D}$ a functor $\hat{F} : C^{op} \times C \longrightarrow \mathcal{D}^{op} \times \mathcal{D}$. We call functors that arise in this way *symmetric* - see [8] for a full definition. Each symmetric functor F induces two functors F_1 and F_2 by post-composition with the projections Π_1 and Π_2 arising from the product on Cat. In fact the mapping $F \mapsto \hat{F}$ is a bijection between difunctors and symmetric functors with inverse sending F to F_2. This fact will be used below to define symmetric functors by giving difunctors. Finally let $\mathcal{P}$ be the category $\mathsf{pCpo}^{op} \times \mathsf{pCpo}$.

With this notation we can give a semantics to types as follows. If a type τ has n-free type variables [3], its interpretation is a symmetric functor

$$[\![\tau]\!] : \mathcal{P}^n \longrightarrow \mathcal{P}$$

Using the bijection mentioned above, we define the symmetric functor $[\![\tau]\!]$ by giving $[\![\tau]\!]_2$. The exceptions to this rule are for the interpretations of

[3] At this point we play a slight price of informality for not indexing judgments by free type variables. However we previously gained by having less notationally cumbersome judgments. We leave the reader to decide if this was an appropriate choice.

recursive types and object types.

$$\begin{array}{lcl}
[\![1]\!]_2 X & = & 1 \\
[\![\tau_1 + \tau_2]\!]_2 X & = & [\![\tau_1]\!]_2 X + [\![\tau_2]\!]_2 X \\
[\![\tau_1 \times \tau_2]\!]_2 X & = & [\![\tau_1]\!]_2 X \otimes [\![\tau_2]\!]_2 X \\
[\![\tau_1 \to \tau_2]\!]_2 X & = & [\![\tau_1]\!]_1 X \rightharpoonup [\![\tau_2]\!]_2 X \\
[\![\mu v.\tau]\!] X & = & ([\![\tau]\!] X)^\dagger
\end{array}$$

where $([\![\tau]\!]X)^\dagger$ is the fixed point of $[\![\tau]\!]X : \mathcal{P} \longrightarrow \mathcal{P}$. Finally, for an object type $\sigma = [l_1 : \tau_1, \ldots, l_m : \tau_m]$, we have

$$[\![[l_1 : \tau_1, \ldots, l_m : \tau_m]]\!] = [\![\mu v.\ v \to \tau_1 \times \cdots \times \tau_m]\!]$$

Unwinding the definition, we thus have

$$[\![\sigma]\!]_2 X \cong [\![\sigma]\!]_2 X \rightharpoonup [\![\tau_1]\!]_2 X \otimes \cdots \otimes [\![\tau_m]\!]_2 X$$

and note that, in this situation, lemma 4.1 applies since we can take F to be the constant functor returning $[\![\tau_1]\!]_2 X \otimes \cdots \otimes [\![\tau_m]\!]_2 X$. Just as we gave an interpretation to types, so we give one to environments. If E is an environment with n-free type variables, then

$$[\![E]\!] : \mathcal{P}^n \longrightarrow \mathcal{P}$$

is the symmetric functor defined by

$$[\![x_1 : \tau_1, \ldots, x_m : \tau_m]\!]_2 X = [\![\tau_1]\!]_2 X \otimes \cdots \otimes [\![\tau_m]\!]_2 X$$

Finally we come to the interpretation for term judgments. If $E \vdash e{:}\tau$ is a judgment using n-type variables, then its interpretation is an indexed family of morphisms

$$[\![E \vdash e{:}\tau]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\tau]\!]_2 A$$

for each symmetric functor $A : \mathcal{P}^n$, i.e. for some $X : \mathsf{pCpo}^n$ the functor A is of the form $A = ((X_1, X_1), \ldots, (X_n, X_n))$. Since the semantic clauses for the term constructs associated with the the basic types $1, +, \times, \to$ are as expected, we leave them as an exercise and focus instead on the judgments for object introduction, update and elimination which we take verbatim from Definition 3.4

- **Object Introduction:** By assumption we are given maps

  $$[\![E, x : \sigma \vdash b_i{:}\tau_i]\!]_A : [\![E, x : \sigma]\!]_2 A \longrightarrow [\![\tau_i]\!]_2 A$$

  in pCpo. Using the definition of $[\![E, x : \sigma]\!]_2$ and the the adjunction between partial product and and partial exponentials, these correspond to the following maps *in the category* Cpo:

$$[\![E]\!]_2 A \longrightarrow ([\![\sigma]\!]_2 A \rightharpoonup [\![\tau_i]\!]_2 A)$$

and hence we get, for each A, one map

$$[\![E]\!]_2 A \longrightarrow ([\![\sigma]\!]_2 A \rightharpoonup [\![\tau_1 \times \cdots \times \tau_n]\!]_2 A)$$

But, since $[\![\sigma]\!]_2 A \rightharpoonup [\![\tau_1 \times \cdots \times \tau_n]\!]_2 A$ is isomorphic to $[\![\sigma]\!]_2 A$, we are done.

- **Object Elimination:** We are given a family of maps

$$[\![E \vdash a{:}\sigma]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\sigma]\!]_2 A$$

and want a map

$$[\![E \vdash a{:}\sigma]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\tau_j]\!]_2 A$$

This can be constructed by postcomposing with the self-application map $[\![\sigma]\!]_2 A \longrightarrow [\![\tau_1]\!]_2 A \times \cdots [\![\tau_n]\!]_2 A$ and then the j'th projection.

- **Object Update:** Start with the map

$$[\![E \vdash a{:}\sigma]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\sigma]\!]_2 A$$

Unwind the isomorphism defining $[\![\sigma]\!]_2 A$. Replace the j'th component of the tuple with

$$[\![E, x : \sigma \vdash b{:}\tau_j]\!]_A : [\![E, x : \sigma]\!]_2 A \longrightarrow [\![\tau_j]\!]_2 A$$

and then refold the isomorphism to get the required map.

5 Wrapper Classes

We saw in the previous section how object types and the associated term judgments can be given a semantics by solving recursive equations of the form $O \cong O \to K$ for some constant K representing the types of the fields of the object type. There is thus an asymmetry in that the self parameter can be consumed by the methods but the methods can't produce new self's or objects. More generally one would like methods to be able to both consume and return the self parameter - this would make sense in both functional and imperative object calculi. Doing this means solving equations of the form

$$O \cong [O, \mathsf{F}\, O]$$

where F is some covariant functor. Such generalised objects are clearly supported by the semantics we have already developed. Also note by instantiating F with the identity functor we get the classic equation $D \cong [D, D]$.

We put this idea to use by asking the following question. Given that both the initial algebra and final coalgebra styles of programming have proven to be very popular in the functional world, can we incorporate them into the world of objects? More precisely, if F is a covariant functor with initial algebra $\mu\mathsf{F}$ and final coalgebra $\nu\mathsf{F}$, can we find an object O which supports the kind of programming enjoyed by $\mu\mathsf{F}$ and $\nu\mathsf{F}$. Of course, since we work in an algebraically compact category $\mu\mathsf{F} = \nu\mathsf{F}$.

We provide a partial positive answer to this question by choosing O to be the fixed point of the equation $O \cong [O, \mathsf{F}\, O]$. Note that our analysis is semantic in that we treat all covariant functors rather than retreating into some restricted syntactic class of functors such as polynomials. For the rest of this section, fix a covariant functor F and define the difunctor $G(X, Y) = X \to \mathsf{F}\, Y$. Also we write *inn* and *out* for the structure maps

$$inn : [O, FO] \longrightarrow O \qquad out : O \longrightarrow [O, FO]$$

of the initial G-dialgebra. Our first result is that objects can be "evaluated" into the final coalgebra and hence enjoy a notion of equality induced by bisimulation.

Lemma 5.1 *O is an F-coalgebra and hence there is a F-coalgebra homomorphism $O \longrightarrow \nu\mathsf{F}$.*

Proof. From lemma 4.1, self application gives a coalgebra $O \longrightarrow FO$. □

Not only is there a map from O to the final F-coalgebra, but also there is a map from the initial algebra to O

Lemma 5.2 *O is an F-algebra and hence there is a F-algebra homomorphism $\mu\mathsf{F} \longrightarrow O$.*

Proof. We would like constructors for O, that is for O to be an F-algebra. Using the isomorphism defining O, the structure map $\mathsf{F}\, O \longrightarrow O$ can be given by a map $\mathsf{F}\, O \longrightarrow [O, \mathsf{F}\, O]$ which we take to be the first projection after uncurrying. Now that O is an F-algebra, the *fold* operation of the initial algebra defines an F-algebra homomorphism $\mu\mathsf{F} \longrightarrow O$. □

That the composite $\mu\mathsf{F} \longrightarrow O \longrightarrow \nu\mathsf{F}$ is the canonical map induced by the initiality of $\mu\mathsf{F}$ and/or the finality of $\nu\mathsf{F}$ relies on the regularity of O. In this setting O is therefore a retract of $\mu\mathsf{F}$ showing it contains the elements of $\mu\mathsf{F}$ but a whole lot more as well.

Next, we wish to consider recursion principles. Initial algebras come with a canonical recursion operator *fold* which arises as the unique map from the initial algebra to some other algebra. Similarly there is a recursion operator *unfold* which arises as the unique map from some coalgebra to the final coalgebra. As we mentioned earlier, O has the universal property of being the initial dialgebra and hence comes with its own recursion principle for defining maps from O to any other dialgebra. Unwinding the definition of dialgebra etc, this gives the principle of *direcursion*.

Definition 5.3 [Direcursion] Let (ϕ, ψ) be a dialgebra with types given in the diagram below. Define $\llparenthesis \phi \rrparenthesis : O \longrightarrow B$ and $[\!(\psi)\!] : A \longrightarrow O$ to be the unique dialgebra homomorphism such that the following diagram commutes:

$$\begin{tikzcd}
{[O, \mathsf{F}\,O]} \arrow[r, "inn_{\mathsf{G}}"] \arrow[d, "\mathsf{G}\,[\!(\psi)\!]\llparenthesis \phi \rrparenthesis"'] & O \arrow[d, "\llparenthesis \phi \rrparenthesis"] \\
{[A, \mathsf{F}\,B]} \arrow[r, "\phi"'] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
O \arrow[r, "out_{\mathsf{G}}"] & {[O, \mathsf{F}\,O]} \\
A \arrow[u, "{[\!(\psi)\!]}"] \arrow[r, "\psi"'] & {[B, \mathsf{F}\,A]} \arrow[u, "\mathsf{G}\,\llparenthesis \phi \rrparenthesis [\!(\psi)\!]"']
\end{tikzcd}$$

By simply chasing the above diagram, one can extract the direcursion principle as two mutually recursive combinators:

Definition 5.4 [Direcursion - combinators]

$$\llparenthesis \phi, \psi \rrparenthesis\, o \triangleq \phi\ ((\mathsf{F}\,\llparenthesis \phi, \psi \rrparenthesis) \circ (out_{\mathsf{G}}\, o) \circ [\!(\phi, \psi)\!])$$

$$[\!(\phi, \psi)\!]\, o \triangleq inn_{\mathsf{G}}\ ((\mathsf{F}\,[\!(\phi, \psi)\!]) \circ (\psi\, o) \circ \llparenthesis \phi, \psi \rrparenthesis)$$

This recursion scheme has been developed as a programming tool by by [7,15] and also opens the way for potential optimisations of based upon fusion, deforestation etc and gives laws for object-oriented programs a la *Algebra of Programming*-school. In future work we plan to test whether thesis is practically viable.

Here, we use direcursion to show that O can be used to simulate the unfold operation of the final F-coalgebra. That is given any F-coalgebra $\alpha : A \to \mathsf{F}\,A$, we define a map from A to O. This can be done by instantiating the direcursion principle by taking B to be the one element cpo. The map ϕ must then be the unique total map, while the map $A \to [1, \mathsf{F}\,A]$ sends a to the total function returning $\alpha(a)$.

To summarise, we have defined a translation of some of the key features of initial algebra and final coalgebra programming into the world of objects.

That is, we have defined an object type which contains the elements of the initial algebra, has constructors for pattern matching, can be evaluated into the final coalgebra, supports a notion of bisimulation and supports an unfold operator. That these constructions are quite simple suggests to us that these wrapper objects are natural and gives us hope that further concepts can be incorporated into the model without it becoming intractable. But that is of course the subject for future research.

6 Conclusion and Further Work

Our approach in this paper differs from the original denotational semantics given in [1]. Firstly and most fundamentally, they use the ideals/metric approach [14] while our approach is based on Fiore's category of partial maps instantiated for pCpo, thus mimicking the more abstract order-enriched setting of [22,2]. Secondly, Abadi and Cardelli interpret types as partial equivalence relations (pers) over a universal domain, while we interpret object types by solving recursive type equations in pCpo. As a result, we get a more intuitive model of objects, with an associated principle of recursion. We think the translation of inductive types into wrappers shows the simplicity and naturalness of this model.

Reus and Streicher [20] have recently treated untyped object calculus in a domain theoretic setting. They use an induction principle to reason with such objects. However, in their work there is one single induction principle, whereas in our typed setting, there is an instantiation for every object type.

An O-categorical model of $\mathbf{F}_{<:\mu}$ poses problems as noted by Abadi and Plotkin [2]. We expect having to address similar problems in the setting of a pCpo model of FPC with subtyping. Nor has this paper have dealt with traits/classes and inheritance. These issues are clearly topics for further work, as is an investigation of imperative features.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martín Abadi and Gordon D. Plotkin. A per model of polymorphism and recursive types. In J. Mitchell, editor, *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 355–365, Philadelphia, 1990. IEEE Computer Society Press.

[3] Kim B. Bruce. Paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206,

April 1994.

[4] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.

[5] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[6] Luca Cardelli. Extensible records in a pure calculus of subtyping. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 373–425. The MIT Press, Cambridge, MA, 1994.

[7] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[8] M. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[9] Marcelo P. Fiore and Gordon D. Plotkin. An extension of models of axiomatic domain theory to models of synthetic domain theory. In *CSL*, pages 129–149, 1996.

[10] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[11] P. Freyd. Algebraically complete categories. In *Proc. 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, 1990.

[12] Peter J. Freyd. Recursive types reduced to inductive types. In *Proceedings 5th IEEE Annual Symp. on Logic in Computer Science, LICS'90*, pages 498–507. IEEE Computer Society Press, June 1990.

[13] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[14] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 165–174. ACM Press, 1984.

[15] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proc. 7th Conf. of Functional Programing Languages and Computer Architecture*, 1995.

[16] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–124. ACM Press, 1990.

[17] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 299–312. ACM Press, 1993.

[18] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, April 1994.

[19] Gordon Plotkin. Pisa notes (on domain theory). Available from http://homepages.inf.ed.ac.uk/gdp/publications.

[20] Bernhard Reus and Thomas Streicher. Semantics and logic of object calculi. In Gordon Plotkin, editor, *Proceedings of the Seventeenth Annual IEEE Symp. on Logic in Computer Science, LICS 2002*. IEEE Computer Society Press, July 2002.

[21] M. Smyth and G. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[22] Michael Smyth and Gordon Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[23] Mitchell Wand. Type inference for objects with instance variables and inheritance. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 97–120. MIT Press, London, 1994.