Computational Soundness and Adequacy for Typed Object Calculus

Johan Glimming
Department for Numerical Analysis and Computer Science
Stockholm University, Sweden
glimming@kth.se

ABSTRACT

By giving a translation from typed object calculus into Plotkin's FPC, we demonstrate that every computationally sound and adequate model of FPC (with eager operational semantics), is also a sound and adequate model of typed object calculus. This establishes that denotational equality is contained in operational equivalence, i.e. that for any such model of typed object calculus, if two terms have equal denotations, then no program (or rather program context) can distinguish between those two terms. Hence we show that FPC models can be used in the study of program transformations (program algebra) for typed object calculus. Our treatment is based on self-application interpretation and subtyping is not considered. The typed object calculus under consideration is a variation of Abadi and Cardelli's first-order calculus with sum and product types, recursive types, and the usual method update and method invocation in a form where the object types have assimilated the recursive types. As an additional result, we prove subject reduction for this calculus.

1. INTRODUCTION

In this paper we will define System $\mathbf{S}^{-}$, a typed object calculus without subtyping, and interpret this calculus into the metalanguage FPC [19, 20] using a self-application encoding, and prove computational soundness and adequacy of $\mathbf{S}^{-}$ with respect to this interpretation. As a corollary, every model of eager FPC that exhibits these two properties, is automatically such a model of typed object calculus. As a consequence, our results make it feasible to use models of FPC as a starting point for the study of semantics of typed object calculus. Given previous research on FPC models, notably by Plotkin and Fiore [19, 6, 8], this formal connection seems to be quite useful (and gives access to a class of computationally adequate models). In particular, program transformations and reasoning principles from FPC models carry over to typed object calculus (notably Freyd's mixed-variant recursion scheme [10]). As an additional result, we establish the subject reduction property of $\mathbf{S}^{-}$, which unlike any calculus studied by Abadi and Cardelli [1] has sum types (and thus can define many standard datatypes more easily).

The importance of computational adequacy is well-known since Plotkin's pioneering work for PCF [21]. Consider a model $\mathcal{M}$ of typed object calculus. Computational adequacy of $\mathcal{M}$ is the property that if a term has a converging (total, not-bottom) denotation, then that term reduces to a value using the rules of the operational semantics (given by a partial function $\rightsquigarrow$). This property is the reverse implication of the following equivalence, the forward direction being known as computational soundness:

$$\exists v\; t \rightsquigarrow v \quad\iff\quad [\![t]\!] \text{ total} \tag{1}$$

Let us say that $t \simeq t'$ whenever for each program context C of ground type we have either that (i) $C[t] \rightsquigarrow v$ and $C[t'] \rightsquigarrow v'$ such that v and v' are equal, or (ii) both $C[t] \Uparrow$ and $C[t'] \Uparrow$ (where $\Uparrow$ means divergence, i.e. $\nexists v\; C[t] \rightsquigarrow v$ and similarly for t') and $C[t'] \Uparrow$. This is operational equivalence relation that Plotkin [21] studied for PCF and, after him, many others for other programming languages. The bi-implication (1) implies the following useful property (assuming that we have soundness, i.e. that $t \rightsquigarrow v$ implies $[\![t]\!] = [\![v]\!]$):

$$[\![t]\!] = [\![t']\!] \quad\Longrightarrow\quad t \simeq t' \tag{2}$$

For a proof, suppose $[\![t]\!] = [\![t']\!]$. It follows by soundness that $[\![C[t]]\!] = [\![C[t']]\!]$ holds for any program context C. Since contexts have ground type, computational adequacy ensures that either $C[t] = C[t']$ or else $C[t]$ and $C[t']$ both diverge. Hence we have $t \simeq t'$. This demonstrates that computational adequacy is a very desirable relationship between operational semantics and denotational semantics, and in particular makes possible to study program transformations (and hence a program algebra) using the model theory of the programming language. This is indeed also the motivation for the present paper.

The idea of using self-application semantics for modelling (or here, interpreting) typed object calculus is not new. It is mentioned by Abadi and Cardelli [1], and appeared already with the work of Kamin [16]. However, recently this approach has been used by Reus, Streicher, and Schwinghammer [23, 22], who have studied it in the context of program logics. We expect our results to be useful for their line of work, although some care must be taken in analysing our computational adequacy result since we are not directly considering any particular model such as theirs.

To the best of our knowledge this is the first full account of these results, and in particular the first proof of computational adequacy of a self-application interpretation with respect to FPC or any of its models. While Viswanathan [25] considers full abstraction for a related typed object calculus, this is not for a self-application interpretation but for a less natural (but interesting, at least from a theoretical viewpoint) interpretation based on a fixed-point operator at the level of terms. Under such an interpretation, it becomes much more complicated to reason with object types, since the universal property associated to recursive types cannot be used directly. Moreover, we demonstrate in this paper that a particular translation must be considered for eager FPC for the results to be attainable. Hence, our detailed account turned out to provide new insights on the relationship between FPC and typed object calculus.

2. TYPED OBJECT CALCULUS WITH RECURSIVE OBJECTS

Abadi and Cardelli have developed a family of object calculi, some of which are more powerful than others, e.g. by having subtyping, recursive types, variance annotations, polymorphism, or *Self*-type in addition to the standard first-order fragment [1]. Table 1 gives an overview of some typed object calculi, including **S** from Abadi and Cardelli's textbook [1] which is particularly expressive, and the simplified calculus $\mathbf{S}^-$ considered in this paper.

Definition 1 Object Calculi

	$\mathbf{FOb_1}$	$\mathbf{FOb}_{1\langle:}$	$\mathbf{FOb}_{1\langle:\mu}$	$\mathbf{S}^-$	$\mathbf{S}$
Subtyping		●	●		●
Recursive types			●	●	●
Obj-binder				●	●
Self-type					●
Variance ann.					●
Products				●	
Coproducts				●	
Functions	●	●	●	●	

Note that $\mathbf{S}^-$ is not a subset of **S**, but contains some extensions such as sum types and functions. These extensions can however also be given to **S** (but Abadi and Cardelli's original presentation of **S** did not include them).

The table shows, for example, that variance annotations are not considered at all in this chapter (this is however no fundamental limitation since such annotations can easily be adjoined to an extended system). $\mathbf{S}^-$ will be defined in detail in the following sections. It is based on the syntax of **S** of [1], but omits the primitive covariant self type and adds some other types instead. A notable omission is subtyping, for which it is currently not known if FPC interpretations (as studied in this paper) can be used. $\mathbf{S}^-$ is essentially a superset of $\mathbf{FOb}_{1\mu}$ of Abadi and Cardelli [1], but with *Obj*-binder instead of the μ-binder, and with the extensions listed in the diagram. Note that we have combined recursive types and object types (using the *Obj* binder) in the sense of **S**, also without the primitive covariant self type. Since $\mathbf{S}^-$ is endowed with products and coproducts, FPC will contain a subset of the rules of $\mathbf{S}^-$.

We will now define $\mathbf{S}^-$ and give some simple examples. We choose n-ary products and coproducts to simplify these examples. We give an operational semantics with a clear notion of values. Our choice of an operational approach permits us to prove computational soundness and adequacy with respect to a denotational model. These results could not be proven were we to have used the reduction based approach as certain reductions are in fact unsound. The reason for this is that a reduction-based semantics admits a degree of non-determinism in evaluations that invalidates the soundness proof. Notably, an object with some terminating methods and some non-terminating methods, is interpreted as a product of functions, such that even the terminating method may become non-terminating under some reduction strategies in FPC.

We assume a countable set of method labels L, type variables V, and term variables U. The types of $\mathbf{S}^-$ are given in definition 2. Notationally, we write $[\ell_i : \tau_i]^{i\in I}$ for $[\ell_1 : \tau_1, ..., \ell_n : \tau_n]$ with $n \in \mathbb{N}$ and equate object types which are equivalent under permutation of the order of labels or under the obvious notion of α-equivalence induced by the type binder *Obj*. We introduce shorthand $\tau_1 \times \tau_2 = \prod_{i\in\{1,2\}} \tau_i$ and similarly $\tau_1 + \tau_2 = \coprod_{i\in\{1,2\}} \tau_i$ for binary products and coproducts.

Definition 2 Syntax of $\mathbf{S}^-$

Syntax for Types

The set $\widehat{\mathcal{T}}$ of pretypes is defined by induction with

τ	::=	X	type
		1	terminal type
		$\prod_{i\in I} \tau_i$	product types
		$\coprod_{i\in I} \tau_i$	coproduct types
		$\tau_1 \to \tau_2$	function types
		$Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$	object types (ℓ_i distinct)

where $X \in V$, and for each i in a finite set I, $\ell_i \in L$ are pairwise distinct.

Syntax for Terms

The set $\widehat{\mathcal{L}}$ of preterms is defined by induction with

m	::=	$\star$	unit
		x_i	term variables
		$\langle m_0, \ldots, m_n \rangle$	tupling
		$\pi_i\ m$	projections
		$case(m_0, x_1.m_1, \ldots, x_n.m_n)$	case
		$\iota_i\ m$	injections
		$m_0\ (m_1)$	λ-application
		$\lambda x : \tau.m$	λ-abstraction
		$Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}$	object introduction
		$m_1.\ell \Leftarrow \varsigma(x : \tau)m_2$	method update
		$m.\ell$	method invocation

where for each $i \in \mathbb{N}$, $x_i \in U$, $X \in V$, $\sigma, \tau_i \in \widehat{\mathcal{T}}$, and for each $i \in I \subseteq \mathbb{N}$ $\ell_i \in L$.

Definition 2 also gives the preterms of $\mathbf{S}^-$. We identify preterms which are equal up to the order of method labels or are equivalent under the obvious notion of α-equivalence induced by the term-binders λ, ς, and *case* and the type binder *Obj*. We use the standard definition of substitution which can be found in Abadi and Cardelli, and write $m\{\!\{a/x\}\!\}$ to mean that a is substituted for all free occurrences of x in m [1]. Further, $m(x)$ means x may occur free in m. We use similar notation for the substitution of types for type variables in both types and terms. When clear from the context, we eliminate the type or term variable being substituted for and simply write $m\{\!\{a\}\!\}$ and $m\{\!\{\tau\}\!\}$.

2.1 Type Theory

A type judgement consists of a sequence of distinct type variables (a type context) together with a type whose free type variables appear in the sequence. The formal definition of type contexts appear in definition 3.

Here are a couple of examples:

Example 2.1. *One may consider representing the Java-like interface*

interface *Point* {**public void** *bump*(); **public int** *val*(); }

as the following type in $\mathbf{S}^-$ *(assuming a type Int exists):*

$$Point = Obj(X)[val : Int, bump : X]$$

Definition 3 Types and type contexts in $\mathbf{S}^{-}$

Type Contexts
Type contexts are generated by the following rules

TyCon Empty
$$\vdash \Diamond$$

TyCon X
$$\frac{\vdash \Theta}{\vdash \langle \Theta, X\rangle} \quad \text{where } X \in V, X \notin \Theta$$

Well-formed Types
The typing judgments $\Theta \vdash \tau$ are those generated by the following rules:

Type X
$$\frac{\vdash \Theta}{\Theta \vdash X} \quad \text{where } X \in \Theta$$

Type Unit
$$\frac{\vdash \Theta}{\Theta \vdash 1}$$

Type Fun
$$\frac{\Theta \vdash \tau_1 \quad \Theta \vdash \tau_2}{\Theta \vdash \tau_1 \to \tau_2}$$

Type Object
$$\frac{\Theta, X \vdash \tau_i \qquad i \in I}{\Theta \vdash Obj(X)[\ell_i : \tau_i(X)]^{i\in I}} \qquad \frac{\Theta \vdash \tau_i \qquad i \in I}{\Theta \vdash \boxdot_{i\in I}\tau_i} \quad \text{where } \boxdot \in \{\prod, \coprod\}$$

We let $\mathcal{T}$ denote the set of well-formed types.

Example 2.2. *The Java-like interface*

interface *UnLam* {**public void** *bump*(); }

gives rise to an object type of the form

$$UnLam = Obj(X)[bump : X]$$

Once we have the type judgements, we can define term contexts (definition 4) and then term judgements (definition 5). As one would expect, terms are closed under substitution. That is, if $\Theta, \langle\Gamma, x : \tau'\rangle \vdash t : \tau$ and $\Theta, \Gamma \vdash t' : \tau'$ are derivable then so is $\Theta, \Gamma \vdash t\{t'/x\} : \tau$.

Convention 2.1. *We say a preterm $m \in \widehat{\mathcal{L}}$ is well-typed if there exists well-formed contexts Θ, Γ and a type judgement $\Theta \vdash \tau$ such that $\Theta, \Gamma \vdash m : \tau$ is derivable. We let $\mathcal{L}$ denote the set of well-typed terms up to α-equivalence and permutations of method labels.*

Example 2.3. *A point whose value is* 0 *and whose bump method adds* 1 *to the value can be represented in* $\mathbf{S}^{-}$ *as*

$$\begin{aligned} p \triangleq \quad & Obj(X = Point)[val = \varsigma(x : X)0, \\ & bump = \varsigma(x : X)x.val \Leftarrow \varsigma(y : X)x.val + 1 \quad] \end{aligned}$$

Unlike Java, $\mathbf{S}^{-}$ makes no distinction between objects and classes. Therefore, a class is represented by an object, which can be cloned or copied into new objects which will (initially at least) have the same methods. There are other differences: object calculus allows methods to be updated, which is impossible in Java, and $\mathbf{S}^{-}$ has no imperative features. Since we have method updates, there is no need to have separate attributes. Attributes, like *val*, are instead identified with method bodies in which the self variable does not occur.

Definition 4 Term contexts for $\mathbf{S}^{-}$

Well-formed term contexts are given by the rules

Con Empty
$$\frac{\vdash \Theta}{\Theta \vdash \Diamond}$$

Con x
$$\frac{\Theta \vdash \tau, \Gamma}{\Theta \vdash \langle \Gamma, x : \tau\rangle} \quad \text{where } x \in U, x \notin \Gamma$$

where $\Diamond$ is the empty sequence.

We recall some standard substitution lemmas (for a proof, see e.g. Barendregt [3] or Sørensen et al [24]), where we use notation *FV* for free variables of a term and similarly *FTV* for type variables (both for terms and types), and write $\equiv$ for syntactical equality.

Lemma 2.1 (Substitution commutativity).

(i) If $X \not\equiv Y$, $X \notin FTV(\tau_2)$, then the following is true:

$$\tau\{\tau_1/X\}\{\tau_2/Y\} \equiv \tau\{\tau_2/Y\}\{\tau_1\{\tau_2/Y\}/X\}$$

(ii) If $x \not\equiv y$, $x \notin FV(t_2)$, then the following is true:

$$t\{t_1/x\}\{t_2/y\} \equiv t\{t_2/y\}\{t_1\{t_2/y\}/x\}$$

(iii) If $X \not\equiv Y$, $X \notin FTV(\tau_2)$, then the following is true:

$$t\{\tau_1/X\}\{\tau_2/Y\} \equiv t\{\tau_2/Y\}\{\tau_1\{\tau_2/Y\}/X\}$$

Lemma 2.2 (Substitutivity). *The following rules are valid:*

type subst
$$\frac{\langle X, \Theta\rangle \vdash \sigma \quad \Theta \vdash \sigma'}{\Theta \vdash \sigma\{\sigma'/X\}}$$

subst term
$$\frac{\Theta, \langle x : \sigma, \Gamma\rangle \vdash t : \sigma' \quad \Theta, \Gamma \vdash s : \sigma}{\Theta, \Gamma \vdash t\{s/x\} : \sigma'}$$

subst type
$$\frac{\langle X, \Theta\rangle, \Gamma \vdash t : \sigma' \quad \Theta \vdash \sigma''}{\Theta, \Gamma \vdash t\{\sigma''/X\} : \sigma'\{\sigma''/X\}}$$

Proof. By induction on the derivation of a well-formed type $\langle X, \Theta\rangle \vdash \sigma$ and well-typed term $\Theta, \langle x : \sigma, \Gamma\rangle \vdash t : \sigma'$ or $\langle X, \Theta\rangle, \Gamma \vdash t : \sigma'$, respectively. For (type subst) and (subst term) this is similar to the theorems given by Abadi et al [1] including for their calculi $\mathbf{Ob}_{1<:\mu}$ and $\mathbf{S}$. The sum type cases are covered as in e.g. Pierce [18] or Sørensen et al [24].

The rule (subst type) is proved by induction on the derivation of $\langle \Theta, X\rangle, \Gamma \vdash t : \sigma'$ for an arbitrary but fixed substitution $\{\sigma''/X\}$ such that $\Theta \vdash \sigma''$. The basis is vacuous since a type substitution acts on term variables as identity. The only cases where free type variables can occur in terms is in an object type σ occurring in a term of the form $Obj(Y = \sigma)[\ell_i = \varsigma(x_i : Y)b_i]^{i\in I}$. For $t = Obj(Y = \sigma)[\ell_i = \varsigma(x_i : Y)b_i]^{i\in I}$ we have either that X is free in σ and thus $X \not\equiv Y$, or else we are done. Suppose $X \not\equiv Y$. Then by induction hypothesis, we have for each $i \in I$ that the following holds:

$$\frac{X, \Theta, \Gamma \vdash b_i\{\sigma/Y\} : \tau_i\{\sigma/Y\} \quad \Theta \vdash \sigma''}{\Theta, \Gamma \vdash b_i\{\sigma/Y\}\{\sigma''/X\} : \tau_i\{\sigma/Y\}\{\sigma''/X\}}$$

Via lemma 2.1 we have that the following rule is also derivable (the condition on the free type variables is ensured by the variable convention):

$$\frac{X, \Theta, \Gamma \vdash b_i\{\sigma/Y\} : \tau_i\{\sigma/Y\} \quad \Theta \vdash \sigma''}{\Theta, \Gamma \vdash b_i\{\sigma''/X\}\{\sigma\{\sigma''/X\}/Y\} : \tau_i\{\sigma''/X\}\{\sigma\{\sigma''/X\}/Y\}} \dagger$$

Note that the premises of these rules must hold by an argument that uses a generation lemma. Moreover, the rule (type subst) gives that $\Theta \vdash \sigma\{\sigma''/X\}$ is an object type. It remains to show that

$$\Theta, \Gamma \vdash t\{\sigma''/X\} : \sigma\{\sigma'/X\}$$

i.e. the conclusion of the (subst type) rule in this case. For this note that the premises for the rule (val object) are precisely the conclusion of the derived rule (†) above, so we are done. The situation for (val update) is similar, and (val select) is trivial. □

Definition 5 Typing judgments for $\mathbf{S}^{-}$

$$\textsc{Val Object}\quad \frac{\sigma \equiv Obj(X)[\ell_i : \tau_i(X)]^{i\in I} \qquad \Theta, \langle\Gamma, x_i : \sigma\rangle \vdash b_i\{\!\{\sigma\}\!\} : \tau_i\{\!\{\sigma\}\!\} \quad \forall i \in I}{\Theta, \Gamma \vdash Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} : \sigma}$$

$$\textsc{Val Select}\quad \frac{o \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} \qquad \Theta, \Gamma \vdash o : \sigma \quad i \in I}{\Theta, \Gamma \vdash o.\ell_i\{\!\{\sigma/X\}\!\} : \tau_i\{\!\{\sigma/X\}\!\}}$$

$$\textsc{Val Update}\quad \frac{\sigma \equiv Obj(X)[\ell_i : \tau_i(X)]^{i\in I} \qquad \Theta, \Gamma \vdash m : \sigma \quad \Theta, \langle\Gamma, x : \sigma\rangle \vdash b\{\!\{\sigma\}\!\} : \tau_j\{\!\{\sigma\}\!\} \quad j \in I}{\Theta, \Gamma \vdash m.\ell_j \Leftarrow \varsigma(x : \sigma)b : \sigma}$$

$$\textsc{Val x}\quad \frac{\Theta \vdash \langle\Gamma, x_i : \tau_i, ...\rangle}{\Theta, \langle\Gamma, x_i : \tau_i, ...\rangle \vdash x_i : \tau_i} \qquad \textsc{Val Unit}\quad \Theta, \Gamma \vdash \star : 1 \qquad \textsc{Val Product}\quad \frac{\Theta, \Gamma \vdash a : \prod_{i\in I} \tau_i \ \ j \in I}{\Theta, \Gamma \vdash \pi_j\, a : \tau_j}$$

$$\textsc{Val Pair}\quad \frac{\Theta, \Gamma \vdash a_1 : \tau_1 \qquad \dots \qquad \Theta, \Gamma \vdash a_n : \tau_n}{\Theta, \Gamma \vdash \langle a_1, \dots, a_n\rangle : \prod_{i\in I} \tau_i} \qquad \textsc{Val Sum}\quad \frac{\Theta, \Gamma \vdash a : \tau_j \ \ j \in I}{\Theta, \Gamma \vdash \iota_i\, a : \coprod_{i\in I} \tau_i}$$

$$\textsc{Val Case}\quad \frac{\Theta, \Gamma \vdash m : \coprod_{i\in I} \sigma_i \quad \Theta, \langle\Gamma, x_j : \sigma_j\rangle \vdash m_j : \tau \quad j \in I}{\Theta, \Gamma \vdash case(m, x_1.m_1, ..., x_n.m_n) : \tau} \qquad \textsc{Val Eval}\quad \frac{\Theta, \Gamma \vdash m_1 : \tau_1 \to \tau_2, m_2 : \tau_1}{\Theta, \Gamma \vdash m_1(m_2) : \tau_2} \qquad \textsc{Val Fun}\quad \frac{\Theta, \langle\Gamma, x : \tau_1\rangle \vdash b : \tau_2}{\Theta, \Gamma \vdash \lambda x : \tau_1.b : \tau_1 \to \tau_2}$$

2.2 Operational Semantics

We have now defined the language of $\mathbf{S}^{-}$, and will give it an operational semantics. The semantics is call-by-value and, in particular, each component of a product must have a value for a projection of the product to attain a value. The rules for the non-object part of the calculus are standard while we feel that those for the introduction, eliminating, and updating objects are reasonable, e.g. one does not reduce under the binder in object intro terms and hence all object intro terms are values. This feeling is reinforced by the results we derive later on soundness and adequacy. The values (or normal/canonical forms) are as follows:

$$\begin{array}{rcl} v & ::= & x \mid 1 \mid \iota_i\, v \mid \langle v_1, \dots, v_n\rangle \mid \lambda x : \tau.m \mid \\ & & Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)m_i]^{i\in I} \end{array}$$

The actual operational rules are given in definition 6. Note that the values are precisely the terms v such that $v \rightsquigarrow v$, where $\rightsquigarrow$ means the reduction relation. This is the statement that values are irreducible in a formal sense. A program p is a term such that for some type τ we have $\vdash p : \tau$, i.e. a well-typed term with empty contexts. The key theorem which means that the implementation of the calculus, as given by its operational semantics, respects compile time type information is the preservation of types as shown in the next theorem.

Theorem 2.1 (Subject Reduction). *If t is a well-typed term* $\Theta, \Gamma \vdash t : \tau$ *such that* $t \rightsquigarrow t'$, *then* $\Theta, \Gamma \vdash t' : \tau$.

Proof. The proof is by induction on the derivation of $t \rightsquigarrow t'$ and is fairly routine. Suppose $\Theta, \Gamma \vdash t : \tau$ and $t \rightsquigarrow t'$. We have omitted trivial cases:

Case (Red Case): We have $t = case(m, x_1.m_1, \dots, x_n.m_n)$ and $\Theta, \Gamma \vdash t : \tau$. Since t is well-typed we have $\Theta, \Gamma \vdash m : \coprod_{i\in I} \sigma_i$ and $\Theta, \langle\Gamma, x : \sigma\rangle \vdash m_i : \tau$ for $i \in I$. We have subderivation $m \rightsquigarrow \iota_k v$ and $m_k\{v/x_k\} \rightsquigarrow t'$. But by induction hypothesis this means, by substitutivity, $t' : \tau$.

Case (Red Product): We have $t = \pi_i(m)$ and $\Theta, \Gamma \vdash t : \tau$, which is to say $m = \langle a_1, \dots, a_n\rangle$ for some $\Theta, \Gamma \vdash a_i : \tau_i$. The result follows by induction hypothesis on the required component.

Case (Red Eval): We have $t = m_1(m_2)$ and $\Theta, \Gamma \vdash t : \tau_2$. Therefore $\Theta, \Gamma \vdash m_1 : \tau_1 \to \tau_2$ and $\Theta, \Gamma \vdash m_2 : \tau_1$. That is to say $m_1 = \lambda x : \tau_2.b$. Now for $m_2 \rightsquigarrow v$ we have $\Theta, \Gamma \vdash b\{v/x\} : \tau_2$ and by induction hypothesis $t' : \tau_2$ as required.

Case (Val Select): we have $t = m.\ell_i$ and $\Theta, \Gamma \vdash t : \tau_i$ for $m = Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}$ and $\Theta, \Gamma \vdash m : \sigma$ with $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$. For $m \rightsquigarrow v'$ and $b_i\{\!\{v', \sigma\}\!\} \rightsquigarrow t'$ we have $\Theta, \Gamma \vdash t' : \tau_i\{\!\{\sigma\}\!\}$ by induction hypothesis.

Case (Val Object): we have $t = m.\ell_j \Leftarrow \varsigma(x : \sigma).b$ and $t : \sigma$. Since $\Theta, \langle\Gamma, x : \sigma\rangle \vdash b_j\{\!\{\sigma\}\!\} : \tau_j\{\!\{\sigma\}\!\}$ we have $\Theta, \Gamma \vdash t' : \sigma$ as required. □

3. FPC

Our intention is to interpret object types as solutions of certain recursive equations. We do this syntactically by translating the object calculus into the FPC. In previous work [11], we have done it also semantically by giving denotational models for the object calculus using some sophisticated categorical model, in which case the equations become domain equations rather than, like here, type equations in the metalanguage FPC.

The target calculus of recursive types is known in the semantics literature as FPC. This system is originally due to Plotkin [19], but detailed expositions are given e.g. by Gunter [14] and Fiore [8]. FPC intuitively arises from $\mathbf{S}^{-}$ by deleting the types and terms related to objects and inserting types and terms related to fixed points of mixed variant type constructors. Thus FPC uses the same countable supplies U and V of type and term variables. We summarise the formal rules in definition 7.

Definition 7 Eager FPC

$$\textsc{Val In}\quad \frac{X \notin \Theta \qquad \Theta, \Gamma \vdash m : \tau\{\!\{\mu X.\tau/X\}\!\}}{\Theta, \Gamma \vdash in_{\mu X.\tau}(m) : \mu X.\tau} \qquad \textsc{Val Out}\quad \frac{\Theta, \Gamma \vdash m : \mu X.\tau}{\Theta, \Gamma \vdash out(m) : \tau\{\!\{\mu X.\tau/X\}\!\}}$$

$$\textsc{Type Rec}\quad \frac{\langle\Theta, X\rangle \vdash \tau}{\Theta \vdash \mu X.\tau} \qquad \textsc{Red Inn}\quad \frac{e \rightsquigarrow v}{in(e) \rightsquigarrow in(v)} \qquad \textsc{Red Out}\quad \frac{e \rightsquigarrow in(v)}{out(e) \rightsquigarrow v}$$

The notions of substitution, α-congruence, contexts, well-formed types, are all identical, except that we replace object type formation with the following rule for well formed recursive types. The preterms of FPC are exactly those of $\mathbf{S}^{-}$, omitting all terms derived from the object formation rule, method updates, and method invocation, and adding to the grammar terms of the form $in_{\mu X.\tau}$ for μ-introduction (Val In) and *out* for μ-elimination (Val Out). The

Definition 6 Operational semantics for $\mathbf{S}^-$

Red x
$$x \rightsquigarrow x$$

Red Unit
$$\star \rightsquigarrow \star$$

Red Pair
$$\frac{m_1 \rightsquigarrow v_1 \quad \ldots \quad m_n \rightsquigarrow v_n}{\langle m_1, \ldots, m_2 \rangle \rightsquigarrow \langle v_1, \ldots, v_n \rangle}$$

Red Proj
$$\frac{m \rightsquigarrow \langle v_1, \ldots, v_n \rangle \qquad 1 \leq i \leq n}{\pi_i(m) \rightsquigarrow v_i}$$

Red Sum
$$\frac{m \rightsquigarrow v}{\iota_j\ m \rightsquigarrow \iota_j\ v}$$

Red Case
$$\frac{m \rightsquigarrow \iota_j(v) \qquad m_j\{\!|v/x_j|\!\} \rightsquigarrow v' \qquad j \in [1, n]}{case(m, x_1.m_1, ..., x_n.m_n) \rightsquigarrow v'}$$

Red Fun
$$\lambda x : \tau.m \rightsquigarrow \lambda x : \tau.m$$

Red Eval
$$\frac{m_1 \rightsquigarrow \lambda x : \tau.b \qquad m_2 \rightsquigarrow v \qquad b\{\!|v/x|\!\} \rightsquigarrow v'}{m_1(m_2) \rightsquigarrow v'}$$

Red Object
$$v \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I}$$
$$v \rightsquigarrow v$$

Red Select
$$\frac{v' \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I} \qquad m \rightsquigarrow v' \quad b_i\{\!|v', \sigma|\!\} \rightsquigarrow v}{m.\ell_i \rightsquigarrow v}$$

Red Update
$$\frac{v \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I} \qquad m \rightsquigarrow v}{m.\ell_j \Leftarrow \varsigma(x : \sigma)b\{\!|\sigma|\!\} \rightsquigarrow Obj(X = \sigma)[\ell_i = \varsigma(x : X)b_i, \ell_j = \varsigma(x : X)b]^{i \in I - \{j\}}}$$

where in the last rule we delete the j'th method from v and then add the updated method $\ell_j = \varsigma(x : X)b$.

term judgments for FPC are similarly obtained from those of $\mathbf{S}^-$, but the Val Object, Val Select and Val Update rules are replaced by two rules for typing recursive types. Finally, the operational semantics of FPC is obtained by deleting Val Object terms as values, removing the operational rules for Red Object, Red Select and Red Update and adding the following values and rules from definition 7 to cope with recursive types.

$$v ::= \ldots \ \mid \ in_{\mu X.\tau}(v)$$

In addition to this *eager* (call-by-value) version of FPC, we will briefly also recall the *lazy* (call-by-name) operational semantics that can be given to this language.

Definition 8 Lazy FPC

Operational Semantics

The following rules take the place of Red Proj, Red Case, and Red Eval and all other rules are the same:

RedL Eval
$$\frac{m_1 \rightsquigarrow \lambda x : \tau.b \quad b\{\!|m_2/x|\!\} \rightsquigarrow v}{m_1(m_2) \rightsquigarrow v}$$

RedL Proj
$$\frac{m \rightsquigarrow \langle m_1, \ldots, m_n \rangle \quad m_i \rightsquigarrow v}{\pi_i(m) \rightsquigarrow v}$$

RedL Case
$$\frac{m \rightsquigarrow \iota_j(k) \quad m_j\{\!|k/x_j|\!\} \rightsquigarrow v' \quad j \in [1, n]}{case(m, x_1.m_1, .., x_n.m_n) \rightsquigarrow v'}$$

Under a lazy semantics, we have more values than we had in the eager semantics. If $t_i, \lambda x.m$ are closed terms, the values now also include:

$$v ::= \ldots \ \ \iota_j\ t \ \mid \ \lambda x.m \ \mid \ \langle t_1, ..., t_n \rangle \ \mid \ in_{\mu X.\tau}(v)$$

4. TRANSLATING OBJECT CALCULUS INTO FPC

This section contain a translation of $\mathbf{S}^-$ into FPC. This translation is at the level of types, terms and operational semantics and we find an excellent fit whereby the operational semantics for FPC is both sound and complete. This allows us to transport the well-understood theory of FPC, in particular its denotational models (e.g. [8, 26, 14]), to $\mathbf{S}^-$.

The encoding of objects uses recursive types contrary to e.g. the recursive record semantics in the literature, e.g. [5, 1]. Notably, the recursive record semantics would give the following interpretation of the $p : Point$ object given in the previous examples:

$p = Y\ \lambda p.\langle 0, \langle \pi_1\ p + 1, \pi_2\ p \rangle \rangle$

where $Y : (\tau \rightarrow \tau) \rightarrow \tau$ is a fixed point combinator (which can be encoded into FPC). The type of p is $\mu X.Int \times X$, but as seen in this example we cannot replace the first component of p without giving a completely new definition of p. We will give p the type $\mu X.(X \rightarrow Int) \times (X \rightarrow X)$. This means that p is denoted simply by a product which enjoys the ordinary projections on each component.

Recall the key feature of the encoding chosen for this work is that it reflects our intuition that the object types of $\mathbf{S}^-$ are fixed points of recursive type equations. More specifically, the recursion is over the self-parameter which occurs both covariantly and contravariantly. This intuition is clearly seen in the Val Object typing rule for $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i \in I}$ which suggests the i'th method will consume the self-parameter, which has type σ, to produce something of type τ_i where σ may occur free, e.g. also be produced. Thus, intuitively, the interpretation of σ should satisfy

$$\lceil \sigma \rceil \cong \lceil \sigma \rceil \rightarrow \lceil \tau_1 \rceil \times \ \cdots \ \times \lceil \sigma \rceil \rightarrow \lceil \tau_n \rceil$$

where, as we mentioned above, each of the τ_i may contain σ. Hence the interpretation of σ should be the fixed point $\mu X.X \rightarrow \lceil \tau_1 \rceil \times \cdots \times X \rightarrow \lceil \tau_n \rceil$ where the τ_i may contain X free. Thus the interpretations of the object types *Point* and *UnLam* are

$$\begin{array}{lcl} \lceil Point \rceil & = & \mu X.(X \rightarrow X) \times (X \rightarrow Int) \\ \lceil UnLam \rceil & = & \mu X.X \rightarrow X \end{array}$$

Note the interpretation of this example shows how the type of untyped lambda terms arises naturally as an object. We do not need to translate type contexts since we have identified the sets of type variables. We thus begin by translating well-formed $\mathbf{S}^-$ types into

FPC-types:

$$\begin{array}{lcl}
\lceil X \rceil & \triangleq & X \\
\lceil 1 \rceil & \triangleq & 1 \\
\lceil A \to B \rceil & \triangleq & \lceil A \rceil \to \lceil B \rceil \\
\lceil \prod_{i \in I} A_i \rceil & \triangleq & \prod_{i \in I} \lceil A_i \rceil \\
\lceil \coprod_{i \in I} A_i \rceil & \triangleq & \coprod_{i \in I} \lceil A_i \rceil
\end{array}$$

As mentioned above, the translation of object types is into the solution of a mixed variance recursive type equation.

$$\lceil Obj(X)[\ell_i : \tau_i(X)]^{i \in I} \rceil \quad \triangleq \quad \mu X. X \to \lceil \tau_1 \rceil \times ... \times X \to \lceil \tau_n \rceil$$

Notice that the translation of types respects substitutions, that is $\lceil \tau[\sigma/X] \rceil = \lceil \tau \rceil[\lceil \sigma \rceil / X]$. We can now syntactically translate (term) contexts:

$$\begin{array}{lcl}
\lceil \Theta \vdash \Diamond \rceil & \triangleq & \Theta \vdash \Diamond \\
\lceil \Theta \vdash \langle \Gamma, x : \tau \rangle \rceil & \triangleq & \Theta \vdash \langle \lceil \Gamma \rceil, x : \lceil \tau \rceil \rangle
\end{array}$$

Now we extend our translation to typing judgments. The translations of terms in the intersection of the calculi are just by induction.

$$\begin{array}{lcl}
\lceil \Theta, \Gamma \vdash x_i : \tau \rceil & \triangleq & \Theta, \lceil \Gamma \rceil \vdash x_i : \lceil \tau \rceil \\
\lceil \Theta, \Gamma \vdash \star : 1 \rceil & \triangleq & \Theta, \lceil \Gamma \rceil \vdash \star : 1 \\
\lceil \Theta, \Gamma \vdash \langle m_0, \ldots, m_n \rangle : \tau \rceil & \triangleq & \Theta, \lceil \Gamma \rceil \vdash \langle \lceil m_0 \rceil, \ldots, \lceil m_n \rceil \rangle : \lceil \tau \rceil \\
\lceil \Theta, \Gamma \vdash \pi_i\, m : \tau \rceil & \triangleq & \Theta, \lceil \Gamma \rceil \vdash \pi_i \lceil m \rceil : \lceil \tau \rceil \\
\lceil \Theta, \Gamma \vdash \iota_j\, m : \tau \rceil & \triangleq & \Theta, \lceil \Gamma \rceil \vdash \iota_j \lceil m \rceil : \lceil \tau \rceil \\
\lceil \Theta, \Gamma \vdash m_0\, m_1 : \tau \rceil & \triangleq & \Theta, \lceil \Gamma \rceil \vdash \lceil m_0 \rceil \lceil m_1 \rceil : \lceil \tau \rceil \\
\lceil \Theta, \Gamma \vdash \lambda(x : \sigma) m : \tau \rceil & \triangleq & \Theta, \lceil \Gamma \rceil \vdash \lambda(x : \lceil \sigma \rceil) \lceil m \rceil : \lceil \tau \rceil
\end{array}$$

$$\begin{array}{l}
\lceil \Theta, \Gamma \vdash case(m_0, x_1.m_1, \ldots, x_n.m_n) : \tau \rceil \\
\quad \triangleq \Theta, \lceil \Gamma \rceil \vdash case(\lceil m_0 \rceil, x_1.\lceil m_1 \rceil, \ldots, x_n.\lceil m_n \rceil) : \lceil \tau \rceil
\end{array}$$

Based on translation of object types, we can translate object introductions, method update, and object elimination (method invocation) in the obvious way:

$$\begin{array}{l}
\lceil \Theta, \Gamma \vdash m : \sigma \rceil \\
\quad \triangleq \Theta, \lceil \Gamma \rceil \vdash in(\langle \lambda x : \lceil \sigma \rceil . \lceil b_1 \{\sigma\} \rceil, ..., \lambda x : \lceil \sigma \rceil . \lceil b_n \{\sigma\} \rceil \rangle) : \lceil \sigma \rceil \\
\lceil \Theta, \Gamma \vdash m.\ell_i \rceil \\
\quad \triangleq \Theta, \lceil \Gamma \rceil \vdash (\pi_i\, \alpha)(\lceil m \rceil) : \lceil \tau_i \{\sigma\} \rceil \\
\lceil \Theta, \Gamma \vdash m.\ell_j \Leftarrow \varsigma(x : \sigma) b\{\sigma\} \rceil \\
\quad \triangleq \Theta, \lceil \Gamma \rceil \vdash in(\langle \pi_1 \alpha, \ldots, \pi_{j-1} \alpha, \lambda x : \lceil \sigma \rceil . \lceil b \rceil \{\sigma\}, \pi_{j+1} \alpha, \ldots, \\
\quad \pi_n \alpha \rangle) : \lceil \sigma \rceil \\
\quad \text{where} \\
\quad \alpha \equiv out(\lceil m \rceil) \\
\quad m \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X) b_i]^{i \in I} \\
\quad \sigma \equiv Obj(X)[\ell_i : \tau_i(X)]^{i \in I}
\end{array}$$

Here π_{l_j} is the projection of a labeled product.

Let F_i be a type expression for each $i \in I$, $I = \{1, ..., n\}$. We have interpreted object types as $\mu X.(X \to F_1\ X) \times ... \times (X \to F_n\ X)$ (where for method invocation, self is applied *after* projection) rather than $\mu X.(X \to F\ X)$ where $F = \prod_{i \in I} F_i$. This is because the latter interpretation would break soundness. Consider, for example the interpretation of method invocation. For soundness, we need to prove that π_{ℓ_j} applied to a term reduces to a value in the case when $m.\ell_j$ reduces to a $\mathbf{S}^-$-value. However, the eager operational semantics of projection in FPC requires that all components of the tuple have a value, and we can easily construct an object for which this would not hold. However, given a lazy operational semantics for FPC (e.g. Winskel [26]) this argument would no longer apply, since partially evaluated terms (in particular products) are included as values.

We will now prove an important lemma which shows that our interpretation function $\lceil - \rceil$ is substitutive on terms (it is trivially substitutive on types):

Lemma 4.1. $\lceil m\{v/x, \sigma/\gamma\} \rceil = \lceil m \rceil \{\lceil v \rceil / x, \lceil \sigma \rceil / \gamma\}$

Proof. The proof is by induction on the image of terms under $\lceil - \rceil$. We need only consider object intro, elim, update under $\lceil - \rceil$:

$$\begin{array}{ll}
 & \lceil Obj(X = \sigma)[\ell_i = \varsigma(x_i : X) b_i]^{i \in I} \{v/x, \sigma/\gamma\} : \delta \rceil \\
= & \text{by definition} \\
 & in(\langle \lambda x : \tau . \lceil b_1 \{\delta, v/x, \sigma/\gamma\} \rceil, ..., \lambda x : \tau . \lceil b_n \{\delta, v/x, \sigma/\gamma\} \rceil \rangle) \\
= & \text{by induction hypothesis on } b_i \\
 & in(\langle \lambda x : \tau . \lceil b_1 \{\delta\} \rceil \{\lceil v \rceil / x, \lceil \sigma \rceil / \gamma\}, ..., \lambda x : \tau . \lceil b_n \{\delta\} \rceil \{\lceil v \rceil / x, \lceil \sigma \rceil / \gamma\} \rangle) \\
= & \text{since } \lceil - \rceil \text{ is substitutive on } in \text{, tupling, and } \lambda \\
 & in(\langle \lambda x : \tau . \lceil b_1 \{\delta\} \rceil, ..., \lambda x : \tau . \lceil b_n \{\delta\} \rceil \rangle) \{\lceil v \rceil / x, \lceil \sigma \rceil / \gamma\} \\
= & \text{by definition} \\
 & \lceil Obj(X = \sigma)[\ell_i = \varsigma(x_i : X) b_i]^{i \in I} \rceil \{\lceil v \rceil / x, \lceil \sigma \rceil / \gamma\}
\end{array}$$

The situation is similar for method invocation and method update, in that $\lceil - \rceil$ will be substitutive on sub-terms formed according to the rules of FPC. □

Our translation preserves types:

Lemma 4.2. *If* $\Theta, \Gamma \vdash t : \tau$ *then* $\lceil \Theta \rceil, \lceil \Gamma \rceil \vdash \lceil t \rceil : \lceil \tau \rceil$

Proof. The proof is by induction on well-typed terms. We consider only Val Object, Val Select, and Val Update, since the other cases follow by induction. Suppose $\Theta, \Gamma \vdash Obj(X = \sigma)[\ell_i = \varsigma(x_i : X) b_i]^{i \in I} : \sigma$ where $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i \in I}$. We must show that $\Theta, \lceil \Gamma \rceil \vdash in(\langle \lambda x : \lceil \sigma \rceil . \lceil b_1 \{\sigma\} \rceil, ..., \lambda x : \lceil \sigma \rceil . \lceil b_n \{\sigma\} \rceil \rangle) : \lceil \sigma \rceil$ where $\lceil \sigma \rceil = \mu X. X \to \lceil \tau_1 \rceil \times ... \times X \to \lceil \tau_n \rceil$.

This follows from the premises of the (μI) rule holds, i.e. if

$$\begin{array}{ll}
\Theta, \lceil \Gamma \rceil \vdash & in(\langle \lambda x : \lceil \sigma \rceil . \lceil b_1 \rceil \{\lceil \sigma \rceil\}, ..., \lambda x : \lceil \sigma \rceil . \lceil b_n \rceil \{\lceil \sigma \rceil\} \rangle) : \\
 & X \to \lceil \tau_1 \rceil \times ... \times X \to \lceil \tau_n \rceil \\
 & \quad \{\mu(X) X \to \lceil \tau_1 \rceil \times ... \times X \to \lceil \tau_n \rceil / X\}
\end{array}$$

The premises of Val Object asserts $\Theta, \langle \Gamma, x_i : \sigma \rangle \vdash b_i\{\sigma\} : \tau_i\{\sigma\}$ which by induction hypothesis means $\Theta, \langle \lceil \Gamma \rceil, x_i : \lceil \sigma \rceil \rangle \vdash \lceil b_i \{\sigma\} \rceil : \lceil \tau_i \{\sigma\} \rceil$. We then have a FPC-term of the required type from the bodies $\lceil b_i \{\sigma\} \rceil = \lceil b_i \rceil \{\lceil \sigma \rceil\}$ by the substitution lemma. The Val Fun and Val Proj rules gives us $\langle \lambda x : X . \lceil b_1 \rceil \{X\}, ..., \lambda x : X . \lceil b_n \rceil \{X\} \rangle \{\lceil \sigma \rceil / X\}$. Finally Val In gives us the required type.

The case for Val Update is almost identical. For Val Select we assume $\Theta, \Gamma \vdash m : \sigma$ where $m = Obj(X = \sigma)[\ell_i = \varsigma(x_i : X) b_i]^{i \in I}$ and $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i \in I}$ and consider $\Theta, \Gamma \vdash m.\ell_i : \tau_i\{\sigma\}$. We want $\Theta, \lceil \Gamma \rceil \vdash (\pi_{l_i}\ out(\lceil m \rceil))(\lceil m \rceil) : \lceil \tau_i \{\sigma\} \rceil$. By induction hypothesis we have $\Theta, \lceil \Gamma \rceil \vdash \lceil m \rceil : \lceil \sigma \rceil$. Further $\Theta, \lceil \Gamma \rceil \vdash out(\lceil m \rceil) : \lceil \sigma \{\mu X. \sigma / X\} \rceil$ and after projection we have the body b_i of type $\tau_i\{\sigma\}$, and the result follows by applying the induction hypothesis to b_i. □

5. SOUNDNESS AND ADEQUACY

We will prove the soundness and adequacy of our translation of $\mathbf{S}^-$ into FPC. This means that the translation of $\mathbf{S}^-$ into FPC is given in such a way that the operational semantics of FPC is strong enough to interpret the operational semantics of $\mathbf{S}^-$ while not being so strong as to give extra computations which were not present in $\mathbf{S}^-$.

We will show that $t \rightsquigarrow v$ implies $\lceil t \rceil \rightsquigarrow \lceil v \rceil$. This establishes that our translation is correct (soundness). We also prove an adequacy result of the operational semantics of $\mathbf{S}^-$. These two results establish that any denotational model of FPC (e.g. those studied by Plotkin and Fiore [6]) is, via the self-application interpretation, a suitable mathematical setting for object calculus. For example, a category such as **pCPO** immediately gives us a denotational model of object calculus (via e.g. [26]).

In what follows we will let $\llbracket - \rrbracket$ denote the interpretation of FPC with respect to some denotational semantics equipped with a notion of totality. Such an interpretation is defined in the usual way by induction on the well-formed types and the well-typed terms, see Fiore and Plotkin [6, 8]. We will speak about a denotation being "total". As in loc. cit. we require that the interpretation is with respect to a category of partial maps [8], viz. $p\mathcal{K}$. A term $\Theta, \Gamma \vdash t : \tau$ is understood to be interpreted by an indexed family of partial maps $\llbracket \Gamma \rrbracket A \rightharpoonup \llbracket \tau \rrbracket A$ (where A ranges over objects in $p\mathcal{K}$). The types and type contexts are interpreted by symmetric functors on the ambient category, i.e. self-dual functors of the form $C^{op} \times C \to C$ (henceforth, $C^{op} \times C$ is written as $\breve{C}$). To summarise, we assume that we have indexed partial maps and (symmetric) functors as follows:

$$\begin{array}{lll} \llbracket \Theta, \Gamma \vdash t : \tau \rrbracket_A & : & \llbracket \Theta \vdash \Gamma \rrbracket (A)_2 \rightharpoonup \llbracket \Theta \vdash \tau \rrbracket (A)_2 \\ \llbracket \Theta \vdash \Gamma \rrbracket & : & p\breve{\mathcal{K}}^{|\Theta|} \to p\breve{\mathcal{K}} \end{array}$$

for $A \in |p\breve{\mathcal{K}}|^{|\Theta|}$ where $|\Theta|$ is the number of type variables appearing in Θ. We wrote $F(X)_2$ for $\Pi_2 \circ F(X)$.

Formally, a partial map is described as a pair $[m, f]$ where m is an admissible mono drawn from a subcategory $\mathcal{D}$ of $\mathcal{K}$. The total maps $[m, f]$ are such partial maps where $m = id$. However, we must identify some representations $[m, f]$ since a single partial map can be written in this form in more than one way. For this, we proceed like Fiore et al [6, 8], and say that partial maps $[m, f]$ and $[n, g]$ describe the same partial map if and only if

$$m = n \circ i \text{ and } f = g \circ i \text{ for some isomorphism } i\,.$$

This immediately gives the formal notion of total maps for $p\mathcal{K}$ as the maps $[id, f]$ up to the equivalence relation induced by such isomorphisms.

We remark that for a domain-theoretic model based on $\mathbf{CPPO}_{\perp!}$ of strict continuous maps and cpos with least elements, totality of a map f is more simply the property that $f(x) = \perp$ implies $x = \perp$ (via the isomorphism between this category and **pCPO** as described e.g. in [8]). Also, for the concrete category **pCPO** of predomains, the usual set-theoretic notion of total partial map can be used, rather than the more general one described above. Moreover, in this case, the notion of partial continuity implies that the domain of definition (admissible monos) must be a Scott-open set [6].

Definition 5.1 (Computational Soundness). *We say that an interpretation $\lceil - \rceil$ of $\mathbf{S}^-$ into FPC is computationally sound if, for every $\Theta, \Gamma \vdash o : \tau$ such that $o \leadsto v$ where v is a value, we have that $\llbracket \lceil \Theta, \Gamma \vdash o : \tau \rceil \rrbracket$ is a total map.*

Definition 5.2 (Computational Adequacy). *We say that an interpretation $\lceil - \rceil$ of $\mathbf{S}^-$ into FPC is computationally adequate if given any $\Theta, \Gamma \vdash o : \tau$, whenever $\llbracket \lceil \Theta, \Gamma \vdash o : \tau \rceil \rrbracket$ is a total map, we also have that $o \leadsto v$ for some value v.*

In order to establish computational soundness and adequacy for the interpretation $\lceil - \rceil$, we require the following theorems. From these, the required result of computational soundness and adequacy follows immediately as a corollary simply by composing the relations appropriately, and by observing that $\lceil v \rceil$ is a value in FPC whenever v is a value in $\mathbf{S}^-$.

Theorem 5.1. *The interpretation $\lceil - \rceil$ has the property that $t \leadsto v$, then $\lceil t \rceil \leadsto \lceil v \rceil$*

Proof. We only check the derivation rules Val Object, Val Select, and Val Update, since the result follows from induction for the other derivation rules. The translation of an Val Object term is an FPC value and hence the theorem holds for terms arising as the result of the Val Object rule.

For Val Select, suppose $m \leadsto v'$ and $b_i\{\!\{v', \sigma\}\!\} \leadsto v$, where $v' = Obj(X = \sigma)[l_1 = \varsigma(x_i : X).b_i]^{i \in I}$. We want to show that $\lceil m.\ell_i \rceil \leadsto \lceil v \rceil$. By induction $\lceil m \rceil \leadsto \lceil v' \rceil$ and hence

$$\pi_i(out\lceil m \rceil) \leadsto \lambda x : \lceil \sigma \rceil . b_i$$

Again, by induction, $\lceil m \rceil \leadsto \lceil v' \rceil$ and $\lceil b_i\{\!\{v', \sigma\}\!\} \rceil \leadsto \lceil v \rceil$. The result then follows by the substitution lemma since $\lceil b_i\{\!\{v', \sigma\}\!\} \rceil = \lceil b_i \rceil \{\!\{\lceil v' \rceil, \sigma\}\!\}$.

For method update, suppose $m \leadsto v$ where $v = Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I}$. In order to prove $\lceil m.\ell_j \Leftarrow \varsigma(x : \sigma).b \rceil \leadsto \lceil v' \rceil$ where $v' = Obj(X = \sigma)[l_i = \varsigma(x : X).b_i, l_j = \varsigma(x : X).b]^{i \in I}$ we must prove that

$$\begin{array}{l} in(\langle \pi_1 \alpha, ..., \lambda x : \lceil \sigma \rceil . \lceil b \rceil, ..., \pi_n \alpha \rangle) \leadsto \\ in(\langle \lambda x : \lceil \sigma \rceil . \lceil b_1 \rceil, ..., \lambda x : \lceil \sigma \rceil . \lceil b \rceil, ..., \lambda x : \lceil \sigma \rceil . \lceil b_n \rceil \rangle) \end{array}$$

where $\alpha = out(\lceil m \rceil)$. By induction

$$\lceil m \rceil \leadsto \lceil v \rceil = in(\langle \lambda x : \lceil \sigma \rceil . \lceil b_1 \rceil, ..., \lambda x : \lceil \sigma \rceil . \lceil b_n \rceil \rangle)$$

and hence $\pi_i \alpha \leadsto \lambda x : \lceil \sigma \rceil . \lceil b_i \rceil$ as required. $\square$

Corollary 5.1. *If an FPC model has the property that $t \leadsto v$ implies $\llbracket t \rrbracket = \llbracket v \rrbracket$, then $\mathbf{S}^-$ has this property for the same model via the translation $\lceil - \rceil$.*

We proceed with adequacy which shows that the operational semantics of FPC is not too strong with respect to the operational semantics of $\mathbf{S}^-$.

Theorem 5.2. *The interpretation $\lceil - \rceil$ has the property that if $\lceil t \rceil \leadsto v$, then there is a v' such that $t \leadsto v'$ and $\lceil v' \rceil = v$*

Proof. The proof is by induction on the derivation tree for $\lceil t \rceil \leadsto v$. If t is a variable or any of the terms related to the standard type constructors of λ-calculus, then the proof is as expected. If t is a Val Object term, then both t and $\lceil t \rceil$ are values and hence the theorem trivially holds. There are two more cases:

If t is given by Val Select, say $t \equiv m.\ell_j$, then $\lceil t \rceil \leadsto v$ must have the following form:

$$\frac{\dfrac{\dfrac{\lceil m \rceil \leadsto in\langle \ldots, \lambda x.b, \ldots \rangle}{out\lceil m \rceil \leadsto \langle \ldots, \lambda x.b, \ldots \rangle}}{\pi_j out\lceil m \rceil \leadsto \lambda x.b} \qquad \lceil m \rceil \leadsto u \qquad \lceil b\{\!\{u/x, \sigma\}\!\} \rceil \leadsto v}{(\pi_j out\lceil m \rceil)(\lceil m \rceil) \leadsto v}$$

We see that the derivation for $\lceil b\{\!\{u/x, \sigma\}\!\} \rceil \leadsto v$ is contained in the above derivation. Therefore we can apply the induction hypothesis to it, and also to the term m. The premises of the rule Val Select are now satisfied, and we can conclude that $t \leadsto \xi$ for value ξ. It remains to be shown that $\lceil \xi \rceil = v$, but this is just the induction hypothesis for $\lceil b\{\!\{u/x\}\!\} \rceil$.

Finally, let $t = m.\ell \Leftarrow \varsigma(x : \sigma)b$ be a method update term given by Val Update, and $\lceil t \rceil \leadsto v$ for some value v. Such a term has the following derivation tree:

$$\frac{\dfrac{\lceil m \rceil \leadsto in\ v}{out\lceil m \rceil \leadsto v \equiv \langle \ldots, \lambda x.b, \ldots \rangle}}{\begin{array}{c} in\langle \pi_1 out\lceil m \rceil, \ldots, \pi_{j-1} out\lceil m \rceil, \\ \lambda x : \lceil \sigma \rceil . \lceil b \rceil \{\!\{\sigma\}\!\}, \pi_{j+1} out\lceil m \rceil, \ldots, \pi_n out\lceil m \rceil \rangle \leadsto v \end{array}}$$

The derivation tree clearly shows that $\lceil t \rceil$ reduces to a value exactly when $\lceil m \rceil$ reduces to a value which means, by induction hypothesis, that we have $m \leadsto u$ for some value u. In other words, the premise of the VAL UPDATE rule is satisfied, so we have indeed that $t \rightsquigarrow u'$ for some value u'. It remains to be shown that $\ulcorner u' \urcorner = v$. However, v has the form indicated in the derivation tree $(\langle \ldots, \lambda x.b, \ldots \rangle)$, which is given as the interpretation of precisely the value $Obj(X = \sigma)[\ell_i = \varsigma(x : X)b_i, l_j = \varsigma(x : X)b]^i \in I$ to which t reduces to by the $(\Leftarrow)$ rule. $\square$

The following main result has now been established (it is a direct consequence of the previous theorems):

COROLLARY 5.2 (MAIN RESULT). *Every computationally sound and adequate model of eager FPC is also such a model of* $\mathbf{S}^-$.

It follows from the proof given by Winskel [26] that the categories **pCPO** and $\mathbf{CPPO}_{\perp!}$ both give computationally sound and adequate models of $\mathbf{S}^-$.

Although adequacy holds, the stronger property of full abstraction does *not* hold for self-application semantics [25]. In order to discuss full abstraction we first need to define a notion of observation equivalence. Following Morris [17], terms are taken to be equivalent if they are equal, in a suitable sense, in all program contexts of a given kind. This can be understood as determining whether two terms behave in the same way, operationally. A notion of contextual equivalence requires (1) another equivalence relation to be chosen and (2) a class of program contexts to be chosen. For typed languages, we can take program contexts of one or more ground type (when such exist). For example, Plotkin [21] used boolean-valued contexts in his pioneering work on PCF. In our case, the following definition can be used instead, where we take $\mathrm{Bool} = 1 + 1$:

DEFINITION 5.3 (CONTEXTUAL EQUIVALENCE [12, 13]). *Say that two closed terms o and o' are* contextually equivalent *(or* operationally congruent*, written $o \cong o'$), if for each closing one-hole contexts $C[-]$ of type* Bool*, we have that $C[o] \rightsquigarrow v$ if and only if $C[o'] \rightsquigarrow v'$. (A context is closing for a term t if $\vdash C[t] : \mathrm{Bool}$.)*

Note that here, we follow Gordon et al and have taken the equivalence (1) to distinguish between terms merely based on their convergence behaviour, and not on whether they reduce to the same values (unlike the equivalence Plotkin studied for PCF).

Gordon et al [12] have demonstrated that this notion of contextual equivalence can be characterised using bisimulations on a canonical labelled transition system (i.e. bisimilarity). The terminology "observational congruence" is justified since Gordon et al proved (using a technique due to Howe [15]) that the equivalence is in fact a congruence relation. Note that this means that coinduction can be used when reasoning with object-based programs, giving an alternative to using a denotational semantics (i.e. to the approach followed in this paper), although without additional structure other than that afforded by a labelled transition system (e.g. fixpoint theorems, Freyd's recursion scheme [10] are not available). At any rate, contextual equivalence makes it possible to consider how closely the operational semantics is connected to a denotational model. Ideally, one would like that programs that behaves the same are exactly those that are denotationally the same, which the following property formalises:

DEFINITION 5.4 (FULL ABSTRACTION). *Full abstraction is the property that for any terms o, o', $[\![\ulcorner \Theta, \Gamma \vdash o : \tau \urcorner]\!] = [\![\ulcorner \Theta, \Gamma \vdash o' : \tau \urcorner]\!]$ if and only if $o \cong o'$, i.e. identified denotations correspond to observationally congruent terms.*

Viswanathan showed that self-application models (such as studied in this paper) cannot have this property [25]. His counterexample is based on defining two terms:

$$\begin{array}{rcl} a & = & \mathrm{Obj}(X = \sigma)[\ell = \varsigma(x : X)x.\ell] \\ b & = & \mathrm{Obj}(X = \sigma)[\ell = \varsigma(x : X)case(x.\ell, y.\iota_1 \star, y.\iota_2 \star)] \end{array}$$

Note that both these terms are typeable with type $\mathrm{Obj}(X)[\ell : 1 + 1]$ (where $1 + 1$ represents a boolean type). Although, $a \cong b$, we do not have equal denotations in any model of eager FPC through our interpretation, since self-application admits application of an object where the l method converges, which gives different function values (see loc. cit.). In retrospect, this is not surprising since there is no way in typed object calculus to observe for a particular method the application of that method to an object where the same method has been updated. Fortunately, the most important direction of the bi-implication is generally regarded to be that denotational equality is included in operational equality, and this is exactly what we have established in the present paper for certain kinds of models.

6. CONCLUSION AND FURTHER WORK

In summary, we have developed an interpretation of $\mathbf{S}^-$, a typed object calculus extended with functions, coproducts and products, into (eager) FPC, and proved computational adequacy and soundness. We have established that models of (eager) FPC, such as Fiore's computationally adequate denotational semantics based on partial maps, or Winskel's exposition of Plotkin's semantics [26], are computationally adequate also for typed object calculus. Since a direct proof of computational adequacy is rather complicated in the presence of recursive object types (compare to the work by Fiore and Plotkin [7]), the use of an interpretation into the metalanguage turned out to simplify matters substantially, while in the end giving the same result.

As is well-known [25], full abstraction does not hold for self-application semantics, although it is more abstract than many other so-called object encodings [4]. However, from a pragmatic point of view, the simplicity of the interpretation is of greater importance than its precise characterisation of objects. The self-application interpretation into FPC that we have studied is simple but comes with a powerful recursion scheme. More precisely, models of FPC studied by Fiore and Plotkin [7, 8] possess a recursion scheme due to Freyd [10]. Using our results, this scheme and results surrounding it carry over to typed object calculus. Hence the current paper has provided a formal connection that can be explored much further.

It is known from work by Fiore and Plotkin [9, 8] that there is a class of enriched categorical model of FPC which are computationally adequate for eager FPC. Notably, these authors gave a precise axiomatisation of a category of partial maps (with order-enrichment), such that a computationally adequate model of FPC arises. Examples included **pCPO** and other **CPO**-categories of partial maps, but also functor categories of **pCPO**, etc. We leave as future work to decide whether such more complex models are needed also for typed object calculus, and if therefore an axiomatic analysis of FPC is called for.

Finally, we would like to remark that subtyping has not been studied in this paper, but is certainly very important and needs to be addressed. To this end, FPC can interpret subtyping using coercion functions, but the details are saved for future work. Here, the work by Abadi and Fiore [2] gives some ideas on how to proceed with such investigations.

Acknowledgments

This work was developed under the stimulating guidance of Neil Ghani, which the author gratefully acknowledges.

7. REFERENCES

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

[2] Martín Abadi and Marcelo Fiore. Syntactic considerations on recursive types. In *Proceedings 11th Annual IEEE Symp. on Logic in Computer Science, LICS'96, New Brunswick, NJ, USA, 27–30 July 1996*, pages 242–252. IEEE Computer Society Press, 1996.

[3] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[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. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, pages 51–67. Springer-Verlag, 1984. Lecture Notes in Computer Science, volume 173.

[6] Marcelo Fiore. Order-enrichment for categories of partial maps. *Mathematical Structures in Computer Science*, 5:533–562, 1995.

[7] Marcelo Fiore and Gordon Plotkin. An axiomatisation of computationally adequate domain theoretic models of FPC. In Samson Abramsky, editor, *Proceedings of the Ninth Annual IEEE Symp. on Logic in Computer Science, LICS 1994*, pages 92–102. IEEE Computer Society Press, July 1994.

[8] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps. Cambridge University Press, Distinguished Dissertations in Computer Science, 1996. 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 *Proceedings of the Computer Science Logic Conf. (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 129–149. Springer-Verlag, 1997.

[10] 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.

[11] Johan Glimming and Neil Ghani. Difunctorial semantics of object calculus. In *Electronic Notes in Theoretical Computer Science*, volume 138. Elsevier, November 2005. Proceedings of the Second Workshop on Object Oriented Developments (WOOD 2004).

[12] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 386–395. ACM Press, 1996.

[13] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. Technical Report 386, Computer Laboratory, University of Cambridge, January 1996.

[14] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[15] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[16] Samuel N. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–87. ACM Press, 1988.

[17] James Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[18] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[19] G. D. Plotkin. A metalanguage for predomains. In *Workshop on the Semantics of Programming Languages*, pages 93–118. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1983.

[20] G. D. Plotkin. Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language and Information, Stanford, 1985.

[21] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[22] Bernhard Reus and Jan Schwinghammer. Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science*, 16(2):313–358, April 2006.

[23] Bernhard Reus and Thomas Streicher. Semantics and logic of object calculi. *Theorertical Computer Science*, 316(1):191–213, 2004.

[24] Morten H. Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149. Elsevier, 2006.

[25] Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science (LICS), 1998)*. IEEE Computer Society, 1998.

[26] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.