Approximation Semantics and Expressive Predicate Assignment for Object-Oriented Programming
(Extended Abstract)

R.N.S. Rowe and S.J. van Bakel
{r.rowe,s.vanbakel}@imperial.ac.uk
Department of Computing, Imperial College London,
180 Queen’s Gate, London SW7 2BZ, UK

Abstract. We consider a semantics for a class-based object-oriented calculus based upon approximation; since in the context of LC such a semantics enjoys a strong correspondence with intersection type assignment systems, we also define such a system for our calculus and show that it is sound and complete. We establish the link with between type (we use the terminology predicate here) assignment and the approximation semantics by showing an approximation result, which leads to a sufficient condition for head-normalisation and termination. We show the expressivity of our predicate system by defining an encoding of Combinatory Logic (and so also LC) into our calculus. We show that this encoding preserves predicate-ability and also that our system characterises the normalising and strongly normalising terms for this encoding, demonstrating that the great analytic capabilities of these predicates can be applied to OO.

1 Introduction

Semantics is a well established area of research for both functional and imperative languages; for the functional programming language side, semantics is mainly denotational, based on Scott’s domain theory [25], whereas for imperative languages it is mainly operational [24]. In this paper, we present the first results of our research in the direction of denotational, type-based semantics for object-oriented (OO) calculi, which we aim to extend towards semantics-based systems of abstract interpretation.

Over the years many expressive type systems have been defined and investigated. Amongst those, the intersection type discipline (ITD) [14, 15, 11, 2] stands out as a system that is closed under β-equality and gives rise to a filter model; it is defined as an extension of Curry’s basic type system for the Lambda Calculus (LC) [10], by allowing term-variables to have many, potentially non-unifiable types. This generalisation leads to a very expressive system: for example, termination (i.e. normalisation) of terms can be characterised by assignable types. Furthermore, intersection type-based models and approximation results show that intersection types describe the full semantical behaviour of typeable terms. Intersection type systems have also been employed successfully in analyses for dead code elimination [17], strictness analysis [20], and control-flow analysis [9], proving them a versatile framework for reasoning about programs. Inspired by this expressive power, investigations have taken place of the suitability of
intersection type assignment for other computational models: for example, van Bakel and Fernández have studied [6, 7] intersection types in the context of Term Rewriting Systems (TRS) and van Bakel studied them in the context of sequent calculi [4].

Also the object-oriented programming paradigm has been the subject of extensive theoretical study over the last two decades. OO languages come in two broad flavours: the object (or prototype) based, and the class based. A number of formal models has been developed [13, 12, 21, 18, 1, 19]; for example, the $\$\text{-}\text{calculus}$ [1] and Featherweight Java (FJ) [19] give elementary models for object based and class-based OO respectively.

In an attempt to bring intersection types to the context of OO, in [5] van Bakel and de’Liguoro presented a system for the $\$\text{-}\text{calculus}$; it sees assignable types as an execution predicate, or applicability predicate, rather than as a functional characterisation as is the view in the context of LC and, as a result, recursive calls are typed individually, with different types. This is also the case in our system.

In the current paper we aim to define type-based semantics for class-based OO, so introduce a notion of intersection type assignment for such languages (we will use the terminology predicates here, to distinguish our notion of types from the traditional notion of class types). In order to be able to concentrate on the essential difficulties, we focus on Featherweight Java [19], a restriction of Java defined by removing all but the most essential features of the full language; Featherweight Java bears a similar relation to Java as LC does to languages such as ML and Haskell; in fact, we will show it to be Turing complete. We will show that the expected properties of a system based on intersection predicates (i.e. soundness and completeness) hold, opening up the possibility to define a predicate-based semantics for FJ. In future work, we will look at adding the normal programming features, and investigate which of the main properties we show in this paper are still achievable.

We also define a notion of approximant for FJ-programs as a finite, rooted segment – that cannot be reduced – of a [head] normal form; we go on to show an approximation result which states that, for every predicate assignable to a term in our system, an approximant of that term exists which can be assigned the same predicate. Interpreting a term by its set of approximants gives an approximation semantics and the approximation result then relates the approximation and the predicate-based semantics. This has, as far as we are aware, not previously been shown for a model of OO. The approximation result allows for a predicate-based analysis of termination.

As is also the case for LC and TRS, in our system this result is shown using a notion of computability; since the notion of reduction we consider is weak, as in [7] to show the approximation result we need to consider a notion of reduction on predicate derivations. We illustrate the expressive power of our calculus by showing that it is Turing complete through an embedding of Combinatory Logic – and thereby also the embedding of LC. We also recall the notion of Curry type assignment, for which we can easily show a principal predicate property and show a predicate preservation result: types assignable to $\lambda$-terms in Curry’s system of simple type assignment correspond to predicates in our system that can be assigned to the interpreted $\lambda$-terms. This is easily extended to the strict intersection type assignment system for LC [2]; this then implies that the collection of predicate-able OO expressions correspond to the $\lambda$-terms that are typeable using intersection types, i.e. all semantically meaningful terms.
In [8] we presented a similar system which here has been simplified. In particular, we have removed the field update feature (which can be modelled using method calls\(^1\)), which gives a more straightforward presentation of system and proofs. We have decoupled our intersection predicate system from the existing class type system, which shows that the approximation result does not depend on the class type system in any way.

For lack of space, proofs are omitted from this paper; we refer the interested reader to http://www.doc.ic.ac.uk/~rnr07 for a version of this paper with detailed proofs.

2 The Calculus FJ\(^ c\)

In this section, we will define our variant of Featherweight Java. It defines classes, which represent abstractions encapsulating both data (stored in fields) and the operations to be performed on that data (encoded as methods). Sharing of behaviour is accomplished through the inheritance of fields and methods from parent classes. Computation is mediated by instances of these classes (called objects), which interact with one another by calling (or invoking) methods and accessing each other’s (or their own) fields. We have removed cast expressions since, as the authors of [19] themselves point out, the presence of downcasts is unsound\(^2\); for this reason we call our calculus FJ\(^ c\). We also leave the constructor method as implicit.

Before defining the calculus itself, we introduce notation to represent and manipulate sequences of entities which we will use in this paper.

**Definition 1 (Sequence Notation).** We use \(\pi(n \in \mathbb{N})\) to represent the list \(1, \ldots, n\). A sequence \(a_1, \ldots, a_n\) is denoted by \(\pi_n\); the subscript can be omitted when the exact number of elements in the sequence is not relevant. We write \(a \in \pi_n\) whenever there exists some \(i \in \{1, \ldots, n\}\) such that \(a = a_i\). The empty sequence is denoted by \(\varepsilon\), and concatenation on sequences by \(\cdot\).

We use familiar meta-variables in our formulation to range over class names (\(C\) and \(D\)), field names (\(f\)), method names (\(m\)) and variables (\(x\)). We distinguish the class name Object (which denotes the root of the class inheritance hierarchy in all programs) and the variable this (which is used to refer to the receiver object in method bodies).

**Definition 2 (FJ\(^ c\) Syntax).** FJ\(^ c\) programs \(P\) consist of a **class table** \(\mathcal{CT}\), comprising the class declarations, and an **expression** \(e\) to be run (corresponding to the body of the main method in a real Java program). They are defined by:

\[
\begin{align*}
e &::= x \mid \text{new } C(\pi) \mid e.f \mid e.m(\pi) \\
fd &::= Cf; \\
md &::= D m(C_1, x_1, \ldots, C_n, x_n) \{ \text{return } e; \} \\
cd &::= \text{class } C \text{ extends } C' \{ \ \bar{fd} \ \bar{md} \} \quad (C \neq \text{Object}) \\
\mathcal{CT} &::= \bar{cd} \\
P &::= (\mathcal{CT}, e)
\end{align*}
\]

\(^1\) We can simulate field update by adding to every class \(C\), for each field \(f_i\) belonging to the class, a method \(C\ \text{update}_{f_i}(x)\) \{ return new \(C(\text{this}.f_1, \ldots, x, \ldots, \text{this}.f_n); \} \).

\(^2\) In the sense that typeable expressions can get stuck at runtime.
From this point, all the concepts defined are program dependent (parametric on the class table); however, since a program is essentially a fixed entity, it will be left as an implicit parameter in the definitions that follow. This is done in the interests of readability, and is a standard simplification in the literature (e.g. [19]). Here, we also point out that we only consider programs which conform to some sensible well-formedness criteria: no cycles in the inheritance hierarchy, and fields and methods in any given branch of the inheritance hierarchy are uniquely named. An exception is made to allow the redeclaration of methods, providing that only the body of the method differs from the previous declaration (in the parlance of class-based OO, this is called method override).

**Definition 3 (Lookup Functions).** The following lookup functions are defined to extract the names of fields and bodies of methods belonging to (and inherited by) a class.

1. The function $F(C)$ returns the list of fields $f_n$ belonging to class $C$ (including those it inherits).
2. The function $Mb(C, m)$ returns a tuple $(\tau, \phi)$, consisting of a sequence of the method $m$'s (as defined in the class $C$) formal parameters and its body.

As usual, substitution is at the basis of reduction in our calculus: when a method is invoked on an object (the receiver) the invocation is replaced by the body of the method that is called, and each of the variables is replaced by a corresponding argument.

**Definition 4 (Reduction).** 1. A term substitution $S = \{ x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \}$ is defined in the standard way, as a total function on expressions that systematically replaces all occurrences of the variables $x_i$ by their corresponding expression $e_i$. We write $e^S$ for $S(e)$.

2. The reduction relation $\rightarrow$ is the smallest relation on expressions satisfying:
   - $\text{new } C((\tau \sigma), f_j) \rightarrow e_j$, for class name $C$ with $F(C) = f_n$ and $j \in \pi$.
   - $\text{new } C((\tau \sigma), m(\sigma), \phi) \rightarrow e^S$, where $S = \{ \text{this } \mapsto \text{new } C(\tau \sigma), x_1 \mapsto \phi_1, \ldots, x_n \mapsto \phi_n \}$, for class name $C$ and method $m$ with $Mb(C, m) = (\tau \sigma, \phi)$.

   and the usual congruence rules for allowing reduction in subexpressions.

3. If $e \rightarrow e'$, then $e$ is the redex and $e'$ the contractum; $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$.

This notion of reduction is confluent.

### 3 Approximation Semantics

In this section we define an approximation semantics for $FJ^F$. The notion of approximant was first introduced in [27] for I.C. Essentially, an approximant is a partially evaluated expression in which the locations of incomplete evaluation (i.e. where reduction may still take place) are explicitly marked by the element $\bot$; thus, they approximate the result of computations. Intuitively, an approximant can be seen as a ‘snapshot’ of a computation, where we focus on that part of the resulting program which will no longer change (i.e. the observable output).
**Definition 5 (Approximants).** 1. The set of *approximants* $\mathcal{FJ}^c$ is defined by the following grammar:

\[
\begin{align*}
a & ::= x \mid \bot \mid a.f \mid a.m(\pi^c) \mid \text{new } C(\pi^c) \quad (n \geq 0) \\
A & ::= x \mid \bot \mid \text{new } C(\overline{\pi}) \quad (n \geq 0) \\
& \quad \mid A.f \mid A.m(\overline{\pi}) \quad (A \neq \bot, A \neq \text{new } C(\overline{\pi}))
\end{align*}
\]

Note that approximate normal forms approximate expressions in (head) normal form. In addition, if we were to extend the notion of reduction so that field accesses and method calls on $\bot$ reduce to $\bot$, then we would find that the approximate normal forms are exactly the normal forms with respect to this extended reduction relation.

The notion of approximation is formalised as follows.

**Definition 6 (Approximation Relation).** The approximation relation $\sqsubseteq$ is the contextual closure of the smallest preorder on approximants satisfying: $\bot \sqsubseteq a$, for all $a$.

The relationship between the approximation relation and reduction is:

**Lemma 7.** If $A \sqsubseteq e$ and $e \rightarrow^* e'$, then $A \sqsubseteq e'$.

Notice that this property expresses that the observable behaviour of a program can only increase (in terms of $\sqsubseteq$) through reduction.

**Definition 8 (Approximants).** The set of *approximants* of $e$ is defined as $\mathcal{A}(e) = \{ A \mid \exists e' \ [e \rightarrow^* e' \ & A \sqsubseteq e'] \}$.

Thus, an approximant (of some expression) is a approximate normal form that approximates some (intermediate) stage of execution. This notion of approximant allows us to define what an approximation model is for $\mathcal{FJ}^c$.

**Definition 9 ($\mathcal{FJ}^c$ Semantics).** An approximation model for an $\mathcal{FJ}^c$ program is a structure $\langle \wp(A), \llbracket \cdot \rrbracket \rangle$, where the interpretation function $\llbracket \cdot \rrbracket$, mapping expressions to elements of the domain, $\wp(A)$, is defined by $\llbracket e \rrbracket = \mathcal{A}(e)$.

As for models of $\text{LC}$, our approximation semantics equates expressions which have the same reduction behaviour, as shown by the following theorem.

**Theorem 10.** $e \rightarrow^* e' \Rightarrow \mathcal{A}(e) = \mathcal{A}(e')$.

### 4 Predicate Assignment

We will now define a notion of predicate assignment which is sound and complete with respect to the approximation semantics defined above in the sense that every predicate assignable to an expression is also assignable to an approximant of that expression, and vice versa. Notice that, since in approximants redexes are replaced by $\bot$, this result is not an immediate consequence of subject reduction; we will see that it is the predicate derivation itself which specifies the approximant in question. This relationship is formalised in the next section.
The predicate assignment system defined below uses intersection predicates; it is influenced by the predicate system for the $\zeta$-calculus as defined in [5], and can ultimately be seen as based upon the strict intersection type system for LC (see [2] for a survey). Our predicates describe the capabilities of an expression (or rather, the object to which that expression evaluates) in terms of (1) the operations that may be performed on it (i.e. accessing a field or invoking a method), and (2) the outcome of performing those operations. In this way, our predicates express detailed properties about the contexts in which expressions can be safely used.

More intuitively, our predicates capture the notion of observational equivalence: two expressions with the same (non-empty) set of assignable predicates will be observationally indistinguishable. Our predicates thus constitute semantic predicates, so for this reason (and also to distinguish them from the already existing Java class types) we do not call them types.

**Definition 11 (Predicates).** The set of predicates (ranged over by $\phi, \psi$) and its subset of strict predicates (ranged over by $\sigma$) are defined by the following grammar (where $\varphi$ ranges over predicate variables, and as for syntax $C$ ranges over class names):

\[
\begin{align*}
\phi, \psi &::= \omega | \sigma | \phi \cap \psi \\
\sigma &::= \varphi | C | \langle f:\sigma \rangle | \langle m:(\phi_1,\ldots,\phi_n) \rightarrow \sigma \rangle & (n \geq 0)
\end{align*}
\]

It is possible to group information stated for an expression in a collection of predicates into intersections from which any specific one can be selected as demanded by the context in which the expression appears. In particular, an intersection may combine different (even non-unifiable) analyses of the same field or method.

Our predicates are strict in the sense of [2] since they must describe the outcome of performing an operation in terms of a(nother) single operation rather than an intersection. We include a predicate constant for each class, which we can use to type objects when a more detailed analysis of the object’s fields and methods is not possible. The predicate constant $\omega$ is a top (maximal) predicate, assignable to all expressions.

**Definition 12 (Subpredicate Relation).** The subpredicate relation $\trianglelefteq$ is the smallest preorder satisfying the following conditions:

\[
\begin{align*}
\phi \trianglelefteq \omega & \quad \text{for all } \phi \\
\phi \trianglelefteq \psi & \quad \text{and } \phi \trianglelefteq \psi' \Rightarrow \phi \trianglelefteq \psi \cap \psi' \\
\phi \cap \psi & \trianglelefteq \psi
\end{align*}
\]

We write $\sim$ for the equivalence relation generated by $\trianglelefteq$, extended by

\[
\begin{align*}
\sigma &\sim \sigma' \Rightarrow \langle f:\sigma \rangle \sim \langle f:\sigma' \rangle \\
\sigma_1 \cap \ldots \cap \sigma_n &\sim \sigma_1' \cap \ldots \cap \sigma_n' \Rightarrow \langle m:(\phi_1,\ldots,\phi_n) \rightarrow \sigma \rangle \sim \langle m:(\phi_1',\ldots,\phi_n') \rightarrow \sigma' \rangle
\end{align*}
\]

We consider predicates modulo $\sim$; in particular, all predicates in an intersection are different and $\omega$ does not appear in an intersection. It is easy to show that $\cap$ is associative, so we write $\sigma_1 \cap \ldots \cap \sigma_n$ (where $n \geq 2$) to denote a general intersection.

---

3 This may be because the object does not contain any fields or methods (as is the case for $\text{Object}$) or more generally because no fields or methods can be safely invoked.
If a method calls itself recursively, this recursive call must be checked, but crucially the same time as typing the method body (so-called late self typing). However, a method body may also make requirements on the receiver, through the use of the variable this. In our system we check that these hold at the same time as typing the method body (so-called early self typing). This checking of requirements on the object itself is where the expressive power of our system resides. If a method calls itself recursively, this recursive call must be checked, but – crucially

Definition 13 (Predicate Environments). 1. A predicate statement is of the form $e : \phi$, where $e$ is called the subject of the statement.
2. An environment $\Pi$ is a set of predicate statements with (distinct) variables as subjects; $\Pi, x : \phi$ stands for $\Pi \cup \{ x : \phi \}$ where $x$ does not appear in $\Pi$.
3. If $\Pi_n$ is a sequence of environments, then $\bigcap \Pi_n$ is the environment defined as follows: $x : \phi_1 \cap \cdots \cap \phi_m \in \bigcap \Pi_n$ if and only if $\{ x : \phi_1, \ldots, x : \phi_m \}$ is the non-empty set of all statements in the union of the environments that have $x$ as the subject.

We will now define our notion of intersection predicate assignment, which is a slight variant of the system defined in [8]:

Definition 14 (Predicate Assignment). Predicate assignment for $FJ^c$ is defined by the natural deduction system given in Fig. 1. The rules in fact operate on the larger set of approximants, but for clarity we abuse notation slightly and use the meta-variable $e$ for expressions rather than $a$. Note that there is no special rule for typing $\bot$, meaning that the only predicate which may be assigned to (a subterm containing) $\bot$ is $\omega$.

The rules of our predicate assignment system are fairly straightforward generalisations of the rules of the strict intersection type assignment system for LC to OO: e.g. (FLD) and (INVK) are analogous to ($\rightarrow E$); (NEWF) and (NEWM) are a form of ($\rightarrow I$); and (OBJ) can be seen as a universal ($\omega$)-like rule for objects only. The only non-standard rule from the point of view of similar work for term rewriting and traditional nominal OO type systems is (NEWM), which derives a predicate for an object that presents an analysis of a method. It makes sense however when viewed as an abstraction introduction rule. Like the corresponding LC typing rule ($\rightarrow I$), the analysis involves typing the body of the abstraction (i.e. the method body), and the assumptions (i.e. requirements) on the formal parameters are encoded in the derived predicate (to be checked on invocation). However, a method body may also make requirements on the receiver, through the use of the variable this. In our system we check that these hold at the same time as typing the method body (so-called early self typing). This checking of requirements on the object itself is where the expressive power of our system resides. If a method calls itself recursively, this recursive call must be checked, but – crucially

\[
\begin{align*}
(\text{VAR}) & : \Pi, x : \phi \vdash x : \phi \quad (\phi \subseteq \sigma) \\
(\text{FLD}) & : \Pi \vdash e : (\langle \sigma \rangle) \\
(\text{JOIN}) & : \Pi \vdash e : \sigma_1 \cap \cdots \cap \sigma_n \\
(\text{NEW}) & : \Pi \vdash e : \phi_1 \cap \cdots \cap \phi_n \\
(\text{INVK}) & : \Pi \vdash e : (\langle \phi_i \cap \cdots \cap \phi_n \rangle) \\
(\text{NEWO}) & : \Pi \vdash e : \phi \\
(\text{NEWF}) & : \Pi \vdash e : \phi_1 \cap \cdots \cap \phi_n \\
(\text{NEWM}) & : \Pi \vdash \text{new } C(\phi_1) : \langle \phi_1 \rangle \\
\end{align*}
\]
– carries a different predicate if a valid derivation is to be found. Thus only recursive calls which terminate at a certain point (i.e. which can be assigned $\omega$, and thus ignored) will be permitted by the system.

As is standard for intersection type assignment systems, our system exhibits both subject reduction and subject expansion; the proof is standard.

**Theorem 15 (Subject reduction and expansion).** Let $e \rightarrow e'$; then $\Pi \vdash e' : \phi$ if and only if $\Pi \vdash e : \phi$.

5 Linking Predicates with Semantics: the Approximation Result

We will now describe the relationship between the predicate system and the approximation semantics, which is expressed through an approximation theorem: this states that for every predicate-able approximant of an expression, the same predicate can be assigned to the expression itself, and vice-versa:

$$\Pi \vdash e : \phi \iff \exists A \in A(e) \Pi \vdash A : \phi$$

As for other systems [3, 7], this result is a direct consequence of the strong normalisability of derivation reduction: the structure of the normal form of a given derivation exactly corresponds to the structure of the approximant. As we see below, this implies that predicate-ability provides a sufficient condition for the (head) normalisation of expressions, i.e. a termination analysis for $\mathcal{E}^f$; it also immediately puts into evidence that predicate assignment is undecidable.

Since reduction on expressions is weak, we need to consider derivation reduction, as in [7]. For lack of space, we will skip the details of this reduction; suffice to say that it is essentially a form of cut-elimination on predicate derivations, defined through the following two basic 'cut' rules:

\[
\begin{array}{c}
\frac{\Pi' \vdash e_1 : \phi_1 \quad \ldots \quad \Pi' \vdash e_n : \phi_n}{\Pi' \vdash \text{new } C(e_n) : \langle f_i : \sigma \rangle}
\end{array}
\]

\[
\begin{array}{c}
\frac{\Pi' \vdash \text{this} : \psi, x_1 : \phi_1, \ldots, x_n : \phi_n}{\Pi' \vdash \text{this} : \sigma}
\end{array}
\]

where $D_b^5$ is the derivation obtained from $D_b$ by replacing all sub-derivations of the form (VAR) :: $\Pi, x; \phi_i \vdash x_i : \sigma$ by (a sub-derivation of $D_i$) $D_i$, and sub-derivations of the form (VAR) :: $\Pi, \text{this} : \psi \vdash \text{this} : \sigma$ by (a sub-derivation of) $D_{self}$. Similarly, $e_b^S$ is the expression obtained from $e_b$ by replacing each variable $x_i$ by the expression $e_i$, and the variable this by new $C(e')$. This reduction creates exactly the derivation for a contractum as suggested by the proof of the subject reduction, but is explicit in all its details, which gives the expressive power to show the approximation result.

\footnote{Note that $\phi_i$ could be an intersection, containing $\sigma$.}
Notice that sub-derivations of the form \( \langle \alpha \rangle :: \Pi \vdash e : \alpha \) do not reduce (although \( e \) might) - they are already in normal form with respect to derivation reduction. This is crucial for the strong normalisation result, since it decouples the reduction of a derivation from the possibly infinite reduction sequence of the expression which it assigns a predicate to.

This notion of derivation reduction is not only sound (i.e. produces valid derivations) but, most importantly, we have that it corresponds to reduction on expressions.

**Theorem 16 (Soundness of Derivation Reduction).** If \( D :: \Pi \vdash e : \phi \) and \( D \to D' \), then \( D' \) is a well-defined derivation, in that there exists some \( e' \) such that \( D' :: \Pi \vdash e' : \phi \), and \( e \to e' \).

The key step in showing the approximation result is proving that this notion of derivation reduction is not only sound but, most importantly, is strongly normalising, and that all valid derivations are computable.

**Definition 17 (Computability).** The set of computable derivations is defined as the smallest set satisfying the following conditions (where \( \text{Comp}(D) \) denotes that \( D \) is a member of the set of computable derivations):

1. \( \text{Comp}(\langle \alpha \rangle :: \Pi \vdash e : \alpha) \).
2. \( \text{Comp}(D :: \Pi \vdash e : \phi) \Leftrightarrow \text{SN}(D :: \Pi \vdash e : \phi) \).
3. \( \text{Comp}(D :: \Pi \vdash e : C) \Leftrightarrow \text{SN}(D :: \Pi \vdash e : C) \).
4. \( \text{Comp}(D :: \Pi \vdash e : (f : \sigma)) \Leftrightarrow \text{Comp}(\langle D, \text{FLD} \rangle :: \Pi \vdash e.f : \sigma) \).
5. \( \text{Comp}(D :: \Pi \vdash e : (m : (\langle n \rangle \to \sigma))) \Leftrightarrow \forall \bar{D}_n [ \forall i \in \bar{\Pi} [ \text{Comp}(D_i :: \Pi \vdash e_i : \phi_i) ] \Rightarrow \text{Comp}(\langle D', \bar{D}_1', \ldots, \bar{D}_n', \text{JOIN} \rangle :: \Pi' \vdash e.m(\bar{n}) : \sigma) ] \)

where \( D' = \langle D' \rangle \subseteq \Pi \) and \( \bar{D}_i = D_i[\Pi' \subseteq \Pi_i] \) for each \( i \in \bar{\Pi} \) with \( \Pi' = \bigcap_i \Pi_i \), and \( D'[\Pi' \subseteq \Pi] \) denotes a derivation of exactly the same shape as \( D \) in which the environment \( \Pi \) is replaced with \( \Pi' \) in each statement of the derivation.

6. \( \text{Comp}(\langle D_1, \ldots, D_m, \text{JOIN} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_m) \Leftrightarrow \forall i \in \bar{\Pi} [ \text{Comp}(D_i) \] .

As can be expected, we show that computable derivations are strongly normalising, and that all valid derivations are computable.

**Theorem 18.**

1. \( \text{Comp}(D :: \Pi \vdash e : \phi) \Rightarrow \text{SN}(D :: \Pi \vdash e : \phi) \).
2. \( D :: \Pi \vdash e : \phi \Rightarrow \text{Comp}(D :: \Pi \vdash e : \phi) \)

Then the key step to the approximation theorem follows directly.

**Theorem 19 (Strong Normalisation).** If \( D :: \Pi \vdash e : \phi \) then \( \text{SN}(D) \).

Finally, the following two properties of approximants and predicate assignment lead to the approximation result itself.

**Lemma 20.**

1. If \( D :: \Pi \vdash a : \phi \) and \( a \subseteq a' \) then there exists a derivation \( D' :: \Pi \vdash a : \phi \).
2. If \( D :: \Pi \vdash e : \phi \) and \( D \) is in normal form with respect to \( \rightarrow_D \), then there exists \( A \) and \( D' \) such that \( A \sqsubseteq e \) and \( D' :: \Pi \vdash A : \phi \).

The first of these two properties simply states the soundness of predicate assignment with respect to the approximation relation. The second is the more interesting, since it expresses the relationship between the structure of a derivation and the approximant. The derivation \( D' \) is constructed from \( D \) by replacing sub-derivations of the form \( \langle \omega \rangle :: \Pi \vdash e : \omega \) by \( \langle \omega \rangle :: \Pi \vdash \bot : \omega \) (thus covering any redexes appearing in \( e \)). Since \( D \) is in normal form, there are also no redexes that carry a non-trivial predicate, ensuring that the expression in the conclusion of \( D' \) is a (normal) approximant. The ‘only if’ part of the approximation result itself then follows easily from the fact that \( \rightarrow_D \) corresponds to reduction of expressions, so \( A \) is also an approximant of \( e \). The ‘if’ part follows from the first property above and subject expansion.

**Theorem 21 (Approximation).** \( \Pi \vdash e : \phi \) iff there exists \( A \in A(e) \) such that \( \Pi \vdash A : \phi \).

In other intersection type systems [3, 7], the approximation theorem underpins characterisation results for various forms of termination. Like the \( \text{LC} \) (and in contrast to the system in [7] for \( \text{TRS} \)) our predicate system gives a full characterisation of normalisability. So predicate-ability gives a guarantee of termination since our normal approximate forms of Definition 5 correspond in structure to expressions in (head) normal form.

**Definition 22 ((Head-)Normal Forms).**

1. The set of expressions in head-normal form (ranged over by \( H \)) is defined by:

\[
H ::= x \mid \text{new } C(\Phi) \mid H.f \mid H.m(\Phi) \ (H \neq \text{new } C(\Phi))
\]

2. The set of expressions in normal form (ranged over by \( N \)) is defined by:

\[
N ::= x \mid \text{new } C(\Phi) \mid N.f \mid N.m(\Phi) \ (N \neq \text{new } C(\Phi))
\]

Notice that the difference between these two notions sits in the second and fourth alternative, where head-normal forms allow arbitrary expressions to be used.

**Lemma 23.**

1. If \( A \neq \bot \) and \( A \sqsubseteq e \), then \( e \) is a head-normal form.
2. If \( A \sqsubseteq e \) and \( A \) does not contain \( \bot \), then \( e \) is a normal form.

Thus any predicate, or, more accurately, any predicate derivation other than those of the form \( (\omega) :: \Pi :: e : \omega \) (which correspond to the approximant \( \bot \)) specifies the structure of a (head) normal form via the normal form of its derivation.

**Definition 24.**

1. A derivation is **strong** if it contains no instances of the rule \( (\omega) \).
2. If the only instances of the \( (\omega) \) rule in a derivation are those typing the arguments to method invocations, then we say it is **\( \omega \)-safe**.
3. For a predicate environment \( \Pi \), if for all \( x:\phi \in \Pi \) either \( \phi = \omega \) or \( \phi \) does not contain \( \omega \) at all, then we say \( \Pi \) is \( \omega \)-safe.

From the approximation result, the following normalisability guarantees are easily achieved.
Theorem 25 (Normalisation).

1. $\Pi \vdash e : \sigma$ if and only if $e$ has a head-normal form.

2. $D :: \Pi \vdash e : \sigma$ with $\omega$-safe $D$ and $\Pi$ only if $e$ has a normal form.

3. $D :: \Pi \vdash e : \sigma$ with $D$ strong if and only if $e$ is strongly normalisable.

Notice that we currently do not have an ‘if and only if’ result for Theorem 25(2), whereas terms with normal forms can be completely characterised in LC. This is because derivation expansion does not preserve $\omega$-safety in general. To see why this is the case consider that while an $\omega$-safe derivation may exist for $\Pi \vdash e_i : \sigma$, no $\omega$-safe derivation may exist for $\Pi \vdash \text{new } C(e_n : \sigma) \cdot f : \sigma$ (due to non-termination in the other expressions $e_j$) even though this expression has the same normal form as $e_i$.

6 Expressivity

In this section we consider the formal expressivity of our OO calculus and predicate system. We show that $\text{FJ}^\ell$ is Turing complete by considering an encoding of Combinatory Logic (CL). Through the approximation result of the previous section all normal forms of the CL program can be assigned a non-trivial predicate in our system. Thus, we have a predicate-based characterisation of all (terminating) computable functions in $\text{OO}$.

Combinatory Logic is a model of computation defined by H. Curry [16] independently of LC. It defines a higher-order term rewriting system over of the function symbols $\{S, K\}$ and the following rewrite rules:

$$
S \ x \ y \ \rightarrow \ x \ (y \ z) \\
K \ x \ y \ \rightarrow \ x
$$

Our encoding of CL in $\text{FJ}^\ell$ is based on a curryfied first-order version of the system above (see [6] for details), where the rules for $S$ and $K$ are expanded so that each new rewrite rule has a single operand, allowing for the partial application of function symbols. Application, the basic engine of reduction in term rewriting systems, is modelled via the invocation of a method named $\text{app}$ belonging to a Combinator interface. Since we do not have interfaces proper in $\text{FJ}^\ell$, we have defined a Combinator class but left the body of the $\text{app}$ method unspecified to indicate that in a full-blown Java program this would be an interface. The reduction rules of curryfied CL each apply to (or are ‘triggered’ by) different ‘versions’ of the $S$ and $K$ combinators; in our encoding these rules are implemented by the bodies of five different versions of the $\text{app}$ method which are each attached to different subclasses (i.e. different versions) of the Combinator class.

Definition 26. The encoding of Combinatory Logic (CL) into the $\text{FJ}^\ell$ program $\text{OOCL}$ (Object-Oriented CL) is defined using the class table in Figure 2 and the function $\llbracket \cdot \rrbracket$ which translates terms of CL into $\text{FJ}^\ell$ expressions, and is defined as follows:

$$
\llbracket x \rrbracket = x \\
\llbracket t_1 t_2 \rrbracket = \llbracket t_1 \rrbracket . \text{app}(\llbracket t_2 \rrbracket) \\
\llbracket K \rrbracket = \text{new } K_1() \\
\llbracket S \rrbracket = \text{new } S_1()
$$

The reduction behaviour of $\text{OOCL}$ mirrors that of CL.

Theorem 27. For CL terms $t_1, t_2$: $t_1 \rightarrow^* t_2$ if and only if $\llbracket t_1 \rrbracket \rightarrow^* \llbracket t_2 \rrbracket$. 

Given the Turing completeness of CL, this result shows that our model of class-based OO is also Turing complete. Although this certainly does not come as a surprise, it is a nice formal property for our calculus to have. In addition, our predicate system can perform the same ‘functional’ analysis as ITS does for LC and CL. This is illustrated by a type preservation result. We focus on Curry’s type system for CL and show we can give equivalent types to OOCL programs.

**Definition 28 (Curry Types).** The set of simple types is defined by the grammar:

\[ \tau ::= \varphi \mid \tau \rightarrow \tau \]

**Definition 29 (Curry Type Assignment for CL).** 1. A basis \( B \) is a set of statements of the form \( x: \tau \) in which each of the variables \( x \) is distinct.

2. Simple types are assigned to CL-term using the following natural deduction system:

\[
\begin{align*}
\text{(VAR):} & \quad B \vdash_{\mathcal{CL}} x: \tau \\
\text{(-E):} & \quad B \vdash_{\mathcal{CL}} \lambda x: \tau . t \rightarrow \tau' \\
\text{(K):} & \quad B \vdash_{\mathcal{CL}} K: \tau \rightarrow \tau' \rightarrow \tau \\
\text{(S):} & \quad B \vdash_{\mathcal{CL}} S: (\tau \rightarrow \tau' \rightarrow \tau'') \rightarrow (\tau \rightarrow \tau') \rightarrow \tau \rightarrow \tau''
\end{align*}
\]

To show type preservation, we first define what the equivalent of Curry’s types are in terms of predicates.

**Definition 30 (Type Translation).** The function \( \llbracket \cdot \rrbracket \), which transforms Curry types into predicates, is defined as follows:

\[
\begin{align*}
\llbracket \varphi \rrbracket & = \varphi \\
\llbracket \tau \rightarrow \tau' \rrbracket & = \langle \text{app} : \llbracket \tau \rrbracket \rightarrow \llbracket \tau' \rrbracket \rangle
\end{align*}
\]

It is extended to bases by: \( \llbracket B \rrbracket = \{ x : \llbracket \tau \rrbracket \mid x : \tau \in B \} \).

\(^5\) Note we have *overloaded* the notation \( \llbracket \cdot \rrbracket \), which we also use for the translation of CL terms to FJ expressions.
We can now show the following type preservation result.

**Theorem 31 (Preservation of Types).** If $B \vdash_{CL} t : \tau$ then $\llbracket B \rrbracket \vdash \llbracket t \rrbracket : \llbracket \tau \rrbracket$.

Furthermore, since the well-known encoding of the LC into CL preserves typeability, we also have a type-preserving encoding of LC into $\text{FJ}^*$; it is straightforward to extend this preservation result to full-blown strict intersection types. We stress that this result really demonstrates the validity of our approach. Indeed, our predicate system actually has more power than intersection type systems for CL, since there not all normal forms are typeable using strict types, whereas in our system they are.

**Lemma 32.** If $\sigma$ is a $\bot$-free approximate normal form of $\text{OOCL}$, then there are $\omega$-safe $\mathcal{D}$ and $\Pi$ and strict predicate $\sigma$ such that $\mathcal{D} : \Pi \vdash e : \sigma$.

Since our system has a subject expansion property (and $\omega$-safe typeability is preserved under expansion for the images of CL terms in $\text{OOCL}$), this leads to a complete characterisation of termination for $\text{OOCL}$.

**Theorem 33.** Let $e$ be an expression such that $e = \llbracket t \rrbracket$ for some CL term $t$; then $e$ has a normal form if and only if there are $\omega$-safe $\mathcal{D}$ and $\Pi$ and strict predicate $\sigma$ such that $\mathcal{D} : \Pi \vdash e : \sigma$.

### 7 Some Observations

In this paper we have shown how the ITD approach can be applied to class-based OO, preserving the main expected properties of intersection type systems. There are however some notable differences between our type system and previous work on LC and TRS upon which our research is based.

Firstly, we point out that when considering the encoding of CL (and via that, LC) in $\text{FJ}^*$, our system provides more than the traditional analysis of terms as functions: there are untypeable LC and CL terms which have typeable images in $\text{OOCL}$. Let $\delta$ be the following CL term: $S (S K K) (S K K)$. Notice that $\delta \delta \rightarrow^* \delta \delta$, i.e. it is unsolvable, and thus can only be given the type $\omega$ (this is also true for $\llbracket \delta \delta \rrbracket$). Now, consider the term $t = S (K \delta) (K \delta)$. Notice that it is a normal form ($\llbracket t \rrbracket$ has a normal form also), but that for any term $t', S (K \delta) (K \delta) t' \rightarrow^* \delta \delta$. In a strict system, no functional analysis is possible for terms since $\sigma \rightarrow \omega$ is not a type and so the only way we can type this term is using $\omega^6$. In our type system however we may assign several forms of predicate to $\llbracket t \rrbracket$. Most simply we can derive $\emptyset \vdash \llbracket t \rrbracket : \delta_3$, but even though a ‘functional’ analysis via the app method is impossible, it is still safe to access the fields of the value resulting from $\llbracket t \rrbracket$ – both $\emptyset \vdash \llbracket t \rrbracket : \langle x : K_2 \rangle$ and $\emptyset \vdash \llbracket t \rrbracket : \langle y : K_2 \rangle$ are also easily derivable statements. In fact, we can derive even more informative types: the expression $\llbracket K \delta \rrbracket$ can be assigned predicates of the form $\sigma_{K_\delta} = \langle \text{app:} (\sigma_1) \rightarrow (\text{app:} (\sigma_2 \cap (\text{app:} (\sigma_2 \rightarrow \sigma_1)) \rightarrow \sigma_1)) \rangle$, and so we can also assign $\langle x : \sigma_{K_\delta} \rangle$ and $\langle y : \sigma_{K_\delta} \rangle$ to $\llbracket t \rrbracket$. Notice that the equivalent $\lambda$-term to $\lambda y. (\lambda x. x x)/(\lambda x. x x)$, which is a weak head normal form without a (head) normal

---

6 In other intersection type systems (e.g. [11]) $\emptyset \rightarrow \omega$ is a permissible type, but is equivalent to $\omega$ (that is $\omega \leq (\emptyset \rightarrow \omega) \leq \omega$) and so semantics based on these type systems identify terms of type $\emptyset \rightarrow \omega$ with unsolvable terms.
form. The ‘functional’ view is that such terms are observationally indistinguishable from unsolvable terms. When encoded in FJ\textsuperscript{\textdagger} however, our type system shows that these terms become meaningful (head-normalisable).

The second observation concerns principal types. In the LC, each normal form has a unique most-specific type: i.e. a type from which all the other assignable types may be generated. This property is important for practical type inference. Our intersection type system for FJ\textsuperscript{\textdagger} does not have such a property. Consider the following program:

```java
class C extends Object {C m() {return new C();}}
```

The expression `new C()` is a normal form, and so we can assign it a non-trivial predicate, but observe that the set of all predicates which may be assigned to this expression is the infinite set \{C, \langle m: () \rightarrow C \rangle, \langle m: () \rightarrow (m: () \rightarrow C) \rangle, \ldots \}. None of these types may be considered the most specific one, since whichever predicate we pick we can always derive a more informative (larger) one. On the one hand, this is exactly what we want: we may make a series of any finite number of calls to the method \(m\) and this is expressed by the predicates. On the other hand, this seems to preclude the possibility of practical type inference for our system. Notice however that these predicates are not unrelated to one another: they each approximate the ‘infinite’ predicate \(\langle m: () \rightarrow (m: () \rightarrow \ldots) \rangle\), which can be finitely represented by the recursive type \(\mu X. (m: () \rightarrow X)\). This type concisely captures the reduction behaviour of `new C()`, showing that when we invoke the method \(m\) on it we again obtain our original term. In LC such families of types arise in connection with fixed point operators. This is not a coincidence: the class \(C\) was recursively defined, and in the face of such self-reference it is not then surprising that this is reflected in our type analysis.

## 8 Conclusions & Future Work

We have considered an approximation-based denotational semantics for class-based OO programs and related this to a predicate-based semantics defined using an intersection type approach. Our work shows that the techniques and strong results of this approach can be transferred straightforwardly from other programming formalisms (i.e. LC and term rewriting systems) to the OO paradigm. Through characterisation results we have shown that our predicate system is powerful enough (at least in principle) to form the basis for expressive analyses of OO programs.

Our work has also highlighted where the OO programming style differs from its functional cousin. In particular we have noted that because of the OO facility for self-reference, it is no longer the case that all normal forms have a most-specific (or principal) type. The types assignable to such normal forms do however seem to be representable using recursive definitions. This observation further motivates and strengthens the case (by no means a new concept in the analysis of OO) for the use of recursive types in this area. Some recent work \[22\] shows that a restricted but still highly expressive form of recursive types can still characterise strongly normalising terms, and we hope to fuse this approach with our own to come to an equally precise but more concise and practical predicate-based treatment of OO.

We would also like to reintroduce more features of full Java back into our calculus, to see if our system can accommodate them whilst maintaining the strong theoretical
properties that we have shown for the core calculus. For example, similar to \( \lambda \mu \) [23], it seems natural to extend our simply typed system to analyse the exception handling features of Java.

References