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 ς-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 ς-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 λ-terms in Curry's system of simple type assignment correspond to predicates in our system that can be assigned to the interpreted λ-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 λ-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$^{¢}$

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$^{¢}$. 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 $\overline{n}$ ($n \in \mathbb{N}$) to represent the list $1,\ldots,n$. A sequence $a_1,\ldots,a_n$ is denoted by $\overrightarrow{a_n}$; the subscript can be omitted when the exact number of elements in the sequence is not relevant. We write $a \in \overrightarrow{a_n}$ whenever there exists some $i \in \{1,\ldots,n\}$ such that $a = a_i$. The empty sequence is denoted by ε, and concatenation on sequences by $\vec{a} \cdot \overrightarrow{a'}$.

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$^{¢}$ Syntax). FJ$^{¢}$ 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:

$$
\begin{array}{lcl}
\mathsf{e} & ::= & x \mid \texttt{new C}(\vec{\mathsf{e}}) \mid \mathsf{e}.f \mid \mathsf{e}.m(\vec{\mathsf{e}}) \\
\mathsf{fd} & ::= & \texttt{C}\,f\texttt{;} \\
\mathsf{md} & ::= & \texttt{D}\ m(\texttt{C}_1\ x_1,\ \ldots,\ \texttt{C}_n\ x_n)\ \texttt{\{ return e; \}} \\
\mathsf{cd} & ::= & \texttt{class C extends C}'\ \texttt{\{}\ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}}\ \texttt{\}} \qquad (\texttt{C} \neq \texttt{Object}) \\
\mathcal{CT} & ::= & \overrightarrow{\mathsf{cd}} \\
P & ::= & (\mathcal{CT}, \mathsf{e})
\end{array}
$$

[1] We can simulate field update by adding to every class C, for each field f_i belonging to the class, a method `C update_`f_i`(x) { return new C(this.`f_1`,...,x,...,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 $\mathcal{F}(\mathrm{C})$ returns the list of fields $\overrightarrow{f_n}$ belonging to class C (including those it inherits).
2. The function $\mathcal{M}b(\mathrm{C},m)$ returns a tuple $(\vec{x},\mathsf{e})$, 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* $\mathsf{S} = \{x_1 \mapsto \mathsf{e}_1, \ldots, x_n \mapsto \mathsf{e}_n\}$ 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 $\mathsf{S}(\mathsf{e})$.

2. The reduction relation $\rightarrow$ is the smallest relation on expressions satisfying:
   - $\mathtt{new\ C}(\ \overrightarrow{\mathsf{e}_n})\,.\,f_j \rightarrow \mathsf{e}_j$, for class name C with $\mathcal{F}(\mathrm{C}) = \overrightarrow{f_n}$ and $j \in \overline{n}$.
   - $\mathtt{new\ C}(\vec{\mathsf{e}})\,.\,m(\overrightarrow{\mathsf{e'}_n}) \rightarrow \mathsf{e}^{\mathsf{S}}$, where $\mathsf{S} = \{\mathtt{this} \mapsto \mathtt{new\ C}(\vec{\mathsf{e}}),\ x_1 \mapsto \mathsf{e'}_1,\ \ldots,\ x_n \mapsto \mathsf{e'}_n\}$, for class name C and method m with $\mathcal{M}b(\mathrm{C},m) = (\overrightarrow{x_n},\mathsf{e})$.

   and the usual congruence rules for allowing reduction in subexpressions.

3. If $\mathsf{e} \rightarrow \mathsf{e'}$, then e is the *redex* and $\mathsf{e'}$ 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$^{\mathfrak{C}}$. The notion of *approximant* was first introduced in [27] for LC. 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* $\text{FJ}^{¢}$ is defined by the following grammar:

$$\begin{array}{rcll} \mathsf{a} & ::= & x \mid \bot \mid \mathsf{a}.f \mid \mathsf{a}.m(\overrightarrow{\mathsf{a}_n}) \mid \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{a}_n}) & (n \geq 0) \\ \mathsf{A} & ::= & x \mid \bot \mid \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{A}_n}) & (n \geq 0) \\ & & \mid \mathsf{A}.f \mid \mathsf{A}.m(\overrightarrow{\mathsf{A}}) & (\mathsf{A} \neq \bot, \mathsf{A} \neq \mathsf{new}\ \mathsf{C}(\overrightarrow{\mathsf{A}_n})) \end{array}$$

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: $\bot \sqsubseteq \mathsf{a}$, for all a.

The relationship between the approximation relation and reduction is:

Lemma 7. *If* $\mathsf{A} \sqsubseteq \mathsf{e}$ *and* $\mathsf{e} \to^* \mathsf{e'}$, *then* $\mathsf{A} \sqsubseteq \mathsf{e'}$.

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 $\mathcal{A}(\mathsf{e}) = \{\, \mathsf{A} \mid \exists\, \mathsf{e'}\ [\mathsf{e} \to^* \mathsf{e'}\ \&\ \mathsf{A} \sqsubseteq \mathsf{e'}] \,\}$.

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 $\text{FJ}^{¢}$.

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

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

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

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 ς-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 ϕ, ψ) and its subset of *strict* predicates (ranged over by σ) are defined by the following grammar (where φ ranges over *predicate variables*, and as for syntax C ranges over class names):

$$\begin{array}{rcl} \phi,\psi & ::= & \omega \mid \sigma \mid \phi \cap \psi \\ \sigma & ::= & \varphi \mid \mathrm{C} \mid \langle f:\sigma \rangle \mid \langle m:(\phi_1,\ldots,\phi_n) \to \sigma \rangle \quad (n \geq 0) \end{array}$$

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[3]. The predicate constant ω is a *top* (maximal) predicate, assignable to all expressions.

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

$$\begin{array}{rcl} \phi & \leqslant & \omega \quad \text{for all } \phi \qquad \phi \cap \psi \leqslant \phi \\ \phi \leqslant \psi \;\&\; \phi \leqslant \psi' & \Rightarrow & \phi \leqslant \psi \cap \psi' \qquad \phi \cap \psi \leqslant \psi \end{array}$$

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

$$\begin{array}{rcl} \sigma \sim \sigma' & \Rightarrow & \langle f:\sigma \rangle \sim \langle f:\sigma' \rangle \\ \forall i \in \overline{n}\,[\phi_i' \sim \phi_i'] \;\&\; \sigma \sim \sigma' & \Rightarrow & \langle m:(\phi_1,\ldots,\phi_n) \to \sigma \rangle \sim \langle m:(\phi_1',\ldots,\phi_n') \to \sigma' \rangle \end{array}$$

We consider predicates modulo $\sim$; in particular, all predicates in an intersection are different and ω 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 `Object`) or more generally because no fields or methods can be safely invoked.

$$(\text{VAR}):\frac{}{\Pi, x{:}\phi \vdash x{:}\sigma}\ (\phi \lhd \sigma) \qquad (\text{FLD}):\frac{\Pi \vdash \mathsf{e}:\langle f:\sigma\rangle}{\Pi \vdash \mathsf{e}.f:\sigma} \qquad (\text{JOIN}):\frac{\Pi \vdash \mathsf{e}:\sigma_1 \ \ldots\ \Pi \vdash \mathsf{e}:\sigma_n}{\Pi \vdash \mathsf{e}:\sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 2)$$

$$(\omega):\frac{}{\Pi \vdash \mathsf{e}:\omega} \qquad (\text{INVK}):\frac{\Pi \vdash \mathsf{e}:\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle \quad \Pi \vdash \mathsf{e}_1:\phi_1 \ \ldots\ \Pi \vdash \mathsf{e}_n:\phi_n}{\Pi \vdash \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}):\sigma}$$

$$(\text{NEWO}):\frac{\Pi \vdash \mathsf{e}_1:\phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n:\phi_n}{\Pi \vdash \mathsf{new}\ \mathrm{C}(\overrightarrow{\mathsf{e}_n}):\mathrm{C}}\ (\mathcal{F}(\mathrm{C}) = \overrightarrow{f_n})$$

$$(\text{NEWF}):\frac{\Pi \vdash \mathsf{e}_1:\phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n:\phi_n}{\Pi \vdash \mathsf{new}\ \mathrm{C}(\overrightarrow{\mathsf{e}_n}):\langle f_i:\sigma\rangle}\ (\mathcal{F}(\mathrm{C}) = \overrightarrow{f_n}, i \in \overline{n}, \phi_i \lhd \sigma, \phi_i \neq \omega)$$

$$(\text{NEWM}):\frac{\{\mathtt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n\} \vdash \mathsf{e}_\mathsf{b}:\sigma \quad \Pi \vdash \mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}}):\psi}{\Pi\ \vdash\ \mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}}):\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle}\ (\mathcal{M}b(\mathrm{C}, m) = (\overrightarrow{x_n}, \mathsf{e}_\mathsf{b}))$$

Fig. 1. Predicate Assignment for FJ$^{\mathfrak{C}}$

Definition 13 (Predicate Environments). 1. A *predicate statement* is of the form $\mathsf{e}{:}\phi$, where e is called the *subject* of the statement.
2. An environment Π 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 Π.
3. If $\overrightarrow{\Pi_n}$ is a sequence of environments, then $\bigcap \overrightarrow{\Pi_n}$ is the environment defined as follows: $x{:}\phi_1 \cap \ldots \cap \phi_m \in \bigcap \overrightarrow{\Pi_n}$ 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$^{\mathfrak{C}}$ 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 ω.

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 analagous to $(\to E)$; (NEWF) and (NEWM) are a form of $(\to I)$; and (OBJ) can be seen as a universal (ω)-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 $(\to I)$, 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

– 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 ω, 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 $\mathsf{e} \to \mathsf{e}'$; then $\Pi \vdash \mathsf{e}' : \phi$ if and only if $\Pi \vdash \mathsf{e} : \phi$.

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: $\Pi \vdash \mathsf{e} : \phi \Leftrightarrow \exists\, \mathsf{A} \in \mathcal{A}(\mathsf{e})\ [\Pi \vdash \mathsf{A} : \phi]$. 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 FJ$^{\text{¢}}$; 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:

$$\dfrac{\dfrac{\boxed{\mathcal{D}_1} \quad \boxed{\mathcal{D}_n}}{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}}{\dfrac{\Pi \vdash \mathsf{new}\ \mathrm{C}(\overrightarrow{\mathsf{e}_n}) : \langle f_i : \sigma \rangle}{\Pi \vdash \mathsf{new}\ \mathrm{C}(\overrightarrow{\mathsf{e}_n}).f_i : \sigma}} \quad \to_{\mathfrak{D}} \quad \boxed{\mathcal{D}_i}\ \Pi \vdash \mathsf{e}_i : \sigma$$

$$\dfrac{\dfrac{\boxed{\mathcal{D}_b} \quad \boxed{\mathcal{D}_{\text{self}}}}{\mathtt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash \mathsf{e}_b : \sigma \quad \Pi \vdash \mathsf{new}\ \mathrm{C}(\overrightarrow{\mathsf{e}'}) : \psi}}{\Pi \vdash \mathsf{new}\ \mathrm{C}(\overrightarrow{\mathsf{e}'}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle \quad \vdots \quad \dfrac{\boxed{\mathcal{D}_1} \quad \boxed{\mathcal{D}_n}}{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}}{\Pi \vdash \mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}}').m(\overrightarrow{\mathsf{e}_n}) : \sigma} \quad \to_{\mathfrak{D}} \quad \boxed{\mathcal{D}_b{}^{S}}\ \Pi \vdash \mathsf{e}_b{}^{\mathsf{S}} : \sigma$$

where $\mathcal{D}_b{}^{S}$ is the derivation obtained from $\mathcal{D}_b$ by replacing all sub-derivations of the form $\langle \text{VAR} \rangle :: \Pi, x_i{:}\phi_i \vdash x_i : \sigma$ by (a sub-derivation of[4]) $\mathcal{D}_i$, and sub-derivations of the form $\langle \text{VAR} \rangle :: \Pi, \mathtt{this}{:}\psi \vdash \mathtt{this} : \sigma$ by (a sub-derivation of) $\mathcal{D}_{\text{self}}$. Similarly, $\mathsf{e}_b{}^{\mathsf{S}}$ is the expression obtained from e_b by replacing each variable x_i by the expression e_i, and the variable $\mathtt{this}$ by $\mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}}')$. 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.

[4] Note that ϕ_i could be an intersection, containing σ.

Notice that sub-derivations of the form $\langle\omega\rangle :: \Pi \vdash \mathsf{e} : \omega$ 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 $\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi$ and $\mathcal{D} \rightarrow_{\mathfrak{D}} \mathcal{D}'$, then $\mathcal{D}'$ is a well-defined derivation, in that there exists some e' such that $\mathcal{D}' :: \Pi \vdash \mathsf{e'} : \phi$, and $\mathsf{e} \rightarrow \mathsf{e'}$.

The key step in showing the approximation result is proving that this notion of derivation reduction is terminating, *i.e. strongly normalising*. In other words, all derivations have a *normal form* with respect to $\rightarrow_{\mathfrak{D}}$. Our proof uses the well-known technique of *computability* [26]; the formal definition of the $\mathit{Comp}(\mathcal{D})$ predicate is, as standard, defined inductively over the structure of predicates:

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

1. $\mathit{Comp}(\langle\omega\rangle :: \Pi \vdash \mathsf{e} : \omega)$.
2. $\mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathsf{e} : \varphi) \Leftrightarrow \mathsf{SN}(\mathcal{D} :: \Pi \vdash \mathsf{e} : \varphi)$.
3. $\mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathsf{e} : \mathrm{C}) \Leftrightarrow \mathsf{SN}(\mathcal{D} :: \Pi \vdash \mathsf{e} : \mathrm{C})$.
4. $\mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathsf{e} : \langle f : \sigma\rangle) \Leftrightarrow \mathit{Comp}(\langle \mathcal{D}, \mathrm{FLD}\rangle :: \Pi \vdash \mathsf{e}.f : \sigma)$.
5. $\mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathsf{e} : \langle m : (\overrightarrow{\phi_n}) \rightarrow \sigma\rangle) \Leftrightarrow$
   $\forall \overrightarrow{\mathcal{D}_n} [\, \forall i \in \overline{n}\ [\mathit{Comp}(\mathcal{D}_i :: \Pi_i \vdash \mathsf{e}_i : \phi_i)] \Rightarrow$
   $\mathit{Comp}(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \mathrm{INVK}\rangle :: \Pi' \vdash \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}) : \sigma)\ ]$
   where $\mathcal{D}' = \mathcal{D}[\Pi' \trianglelefteq \Pi]$ and $\mathcal{D}'_i = \mathcal{D}_i[\Pi' \trianglelefteq \Pi_i]$ for each $i \in \overline{n}$ with $\Pi' = \bigcap \Pi \cdot \overrightarrow{\Pi_n}$, and $\mathcal{D}[\Pi' \trianglelefteq \Pi]$ denotes a derivation of exactly the same shape as $\mathcal{D}$ in which the environment Π is replaced with Π' in each statement of the derivation.
6. $\mathit{Comp}(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \mathrm{JOIN}\rangle :: \Pi \vdash \mathsf{e} : \sigma_1 \cap \ldots \cap \sigma_n) \Leftrightarrow \forall i \in \overline{n}\ [\mathit{Comp}(\mathcal{D}_i)]$.

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

Theorem 18. *1.* $\mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi) \Rightarrow \mathsf{SN}(\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi)$.
2. $\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi \Rightarrow \mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi)$

Then the key step to the approximation theorem follows directly.

Theorem 19 (Strong Normalisation). If $\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi$ then $\mathsf{SN}(\mathcal{D})$.

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

Lemma 20. *1. If* $\mathcal{D} :: \Pi \vdash \mathsf{a} : \phi$ *and* $\mathsf{a} \sqsubseteq \mathsf{a'}$ *then there exists a derivation* $\mathcal{D}' :: \Pi \vdash \mathsf{a'} : \phi$.

2. *If* $\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi$ *and* $\mathcal{D}$ *is in normal form with respect to* $\rightarrow_{\mathfrak{D}}$*, then there exists* A *and* $\mathcal{D}'$ *such that* $\mathsf{A} \sqsubseteq \mathsf{e}$ *and* $\mathcal{D}' :: \Pi \vdash \mathsf{A} : \phi$.

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 $\mathcal{D}'$ is constructed from $\mathcal{D}$ by replacing sub-derivations of the form $\langle\omega\rangle :: \Pi \vdash \mathsf{e} : \omega$ by $\langle\omega\rangle :: \Pi \vdash \bot : \omega$ (thus covering any redexes appearing in e). Since $\mathcal{D}$ is in normal form, there are also no redexes that carry a non-trivial predicate, ensuring that the expression in the conclusion of $\mathcal{D}'$ is a (normal) approximant. The 'only if' part of the approximation result itself then follows easily from the fact that $\rightarrow_{\mathfrak{D}}$ 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). $\Pi \vdash \mathsf{e} : \phi$ iff there exists $\mathsf{A} \in \mathcal{A}(\mathsf{e})$ such that $\Pi \vdash \mathsf{A} : \phi$.

In other intersection type systems [3, 7], the approximation theorem underpins characterisation results for various forms of termination. Like the LC (and in contrast to the system in [7] for 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:

$$\mathsf{H} \quad ::= \quad x \mid \mathsf{new}\ \mathsf{C}(\vec{\mathsf{e}}) \mid \mathsf{H}.f \mid \mathsf{H}.m(\vec{\mathsf{e}}) \quad (\mathsf{H} \neq \mathsf{new}\ \mathsf{C}(\vec{\mathsf{e}}))$$

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

$$\mathsf{N} \quad ::= \quad x \mid \mathsf{new}\ \mathsf{C}(\vec{\mathsf{N}}) \mid \mathsf{N}.f \mid \mathsf{N}.m(\vec{\mathsf{N}}) \quad (\mathsf{N} \neq \mathsf{new}\ \mathsf{C}(\vec{\mathsf{N}}))$$

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* $\mathsf{A} \neq \bot$ *and* $\mathsf{A} \sqsubseteq \mathsf{e}$*, then* e *is a head-normal form.*
2. If $\mathsf{A} \sqsubseteq \mathsf{e}$ *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 $\langle\omega\rangle :: \Pi \vdash \mathsf{e} : \omega$ (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 (ω).
2. If the only instances of the (ω) rule in a derivation are those typing the arguments to method invocations, then we say it is ω-*safe* .
3. For a predicate environment Π, if for all $x{:}\phi \in \Pi$ either $\phi = \omega$ or ϕ does not contain ω at all, then we say Π is ω-safe.

From the approximation result, the following normalisability guarantees are easily achieved.

Theorem 25 (Normalisation). 1. $\Pi \vdash \mathsf{e} : \sigma$ if and only if e has a head-normal form.
2. $\mathcal{D} :: \Pi \vdash \mathsf{e} : \sigma$ with ω-safe $\mathcal{D}$ and Π only if e has a normal form.
3. $\mathcal{D} :: \Pi \vdash \mathsf{e} : \sigma$ with $\mathcal{D}$ 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 ω-safety in general. To see why this is the case consider that while an ω-safe derivation may exist for $\Pi \vdash \mathsf{e}_i : \sigma$, no ω-safe derivation may exist for $\Pi \vdash \mathtt{new\ C}(\overrightarrow{\mathsf{e}_n})\,.\,f_i : \sigma$ (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 FJ$^{\mathfrak{C}}$ 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 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 $\{\mathbf{S}, \mathbf{K}\}$ and the following rewrite rules:

$$\begin{array}{rcl} \mathbf{K}\,x\,y & \to & x \\ \mathbf{S}\,x\,y\,z & \to & x\,z\,(y\,z) \end{array}$$

Our encoding of CL in FJ$^{\mathfrak{C}}$ 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 `app` belonging to a `Combinator` *interface*. Since we do not have interfaces proper in FJ$^{\mathfrak{C}}$, we have defined a `Combinator` class but left the body of the `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 `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 FJ$^{\mathfrak{C}}$ program OOCL (Object-Oriented CL) is defined using the class table in Figure 2 and the function $[\![\cdot]\!]$ which translates terms of CL into FJ$^{\mathfrak{C}}$ expressions, and is defined as follows:

$$\begin{array}{rclcrcl} [\![x]\!] & = & x & & [\![t_1 t_2]\!] & = & [\![t_1]\!]\,.\,\mathtt{app}([\![t_2]\!]) \\ [\![\mathbf{K}]\!] & = & \mathtt{new\ K_1()} & & [\![\mathbf{S}]\!] & = & \mathtt{new\ S_1()} \end{array}$$

The reduction behaviour of OOCL mirrors that of CL.

Theorem 27. *For* CL *terms* t_1, t_2*:* $t_1 \to^* t_2$ *if and only if* $[\![t_1]\!] \to^* [\![t_2]\!]$.

```
class Combinator extends Object {
      Combinator app(Combinator x) { return this; } }
class K1 extends Combinator {
      Combinator app(Combinator x) { return new K2(x); } }
class K2 extends K1 { Combinator x;
      Combinator app(Combinator y) { return this.x; } }
class S1 extends Combinator {
      Combinator app(Combinator x) { return new S2(x); } }
class S2 extends S1 { Combinator x;
      Combinator app(Combinator y) { return new S3(this.x, y); } }
class S3 extends S2 { Combinator y;
      Combinator app(Combinator z) {
          return this.x.app(z).app(this.y.app(z)); } }
```

Fig. 2. The class table for Object-Oriented Combinatory Logic (OOCL) programs

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 ITD 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:

$$\tau \;::=\; \varphi \;\mid\; \tau \to \tau$$

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:

$$(\text{VAR}):\ \frac{}{\mathsf{B} \vdash_{\overline{\text{CL}}} x{:}\tau}\ (x{:}\tau \in \mathsf{B}) \qquad (\to\text{E}):\ \frac{\mathsf{B} \vdash_{\overline{\text{CL}}} t_1{:}\tau \to \tau' \quad \mathsf{B} \vdash_{\overline{\text{CL}}} t_2{:}\tau}{\mathsf{B} \vdash_{\overline{\text{CL}}} t_1 t_2{:}\tau'}$$

$$(\mathbf{K}):\ \frac{}{\mathsf{B} \vdash_{\overline{\text{CL}}} \mathbf{K}{:}\tau \to \tau' \to \tau} \qquad (\mathbf{S}):\ \frac{}{\mathsf{B} \vdash_{\overline{\text{CL}}} \mathbf{S}{:}(\tau \to \tau' \to \tau'') \to (\tau \to \tau') \to \tau \to \tau''}$$

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 $[\![\cdot]\!]$, which transforms Curry types into predicates[5], is defined as follows:

$$\begin{array}{rcl} [\![\varphi]\!] & = & \varphi \\ [\![\tau \to \tau']\!] & = & \langle \mathsf{app} : [\![\tau]\!] \to [\![\tau']\!] \rangle \end{array}$$

It is extended to bases by: $[\![\mathsf{B}]\!] = \{x{:}[\![\tau]\!] \mid x{:}\tau \in \mathsf{B}\}$.

[5] Note we have *overloaded* the notation $[\![\cdot]\!]$, which we also use for the translation of CL terms to $\text{FJ}^{\mathfrak{C}}$ expressions.

We can now show the following type preservation result.

Theorem 31 (Preservation of Types). If $\mathsf{B} \vdash_{\overline{\text{CL}}} t{:}\tau$ then $[\![\mathsf{B}]\!] \vdash [\![t]\!] : [\![\tau]\!]$.

Furthermore, since the well-known encoding of the LC into CL preserves typeability, we also have a type-preserving encoding of LC into 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* e *is a* $\bot$*-free approximate normal form of* OOCL*, then there are* ω*-safe* $\mathcal{D}$ *and* Π *and strict predicate* σ *such that* $\mathcal{D} :: \Pi \vdash \mathsf{e} : \sigma$.

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

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

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 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 OOCL. Let δ be the following CL term: $\mathbf{S}\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})$. Notice that $\delta\ \delta \to^* \delta\ \delta$, *i.e.* it is unsolvable, and thus can only be given the type ω (this is also true for $[\![\delta\ \delta]\!]$). Now, consider the term $t = \mathbf{S}\ (\mathbf{K}\ \delta)\ (\mathbf{K}\ \delta)$. Notice that it is a normal form ($[\![t]\!]$ has a normal form also), but that for any term t', $\mathbf{S}\ (\mathbf{K}\ \delta)\ (\mathbf{K}\ \delta)\ t' \to^* \delta\ \delta$. In a strict system, no functional analysis is possible for tsince $\phi \to \omega$ is not a type and so the only way we can type this term is using ω[6]. In our type system however we may assign several forms of predicate to $[\![t]\!]$. Most simply we can derive $\emptyset \vdash [\![t]\!] : \mathsf{S}_3$, 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 $[\![t]\!]$ – both $\emptyset \vdash [\![t]\!] : \langle \mathsf{x} : \mathsf{K}_2 \rangle$ and $\emptyset \vdash [\![t]\!] : \langle \mathsf{y} : \mathsf{K}_2 \rangle$ are also easily derivable statements. In fact, we can derive even more informative types: the expression $[\![\mathbf{K}\ \delta]\!]$ can be assigned predicates of the form $\sigma_{\mathbf{K}\delta} = \langle \mathsf{app} : (\sigma_1) \to \langle \mathsf{app} : (\sigma_2 \cap \langle \mathsf{app} : (\sigma_2) \to \sigma_3 \rangle) \to \sigma_3 \rangle \rangle$, and so we can also assign $\langle \mathsf{x} : \sigma_{\mathbf{K}\delta} \rangle$ and $\langle \mathsf{y} : \sigma_{\mathbf{K}\delta} \rangle$ to $[\![t]\!]$. Notice that the equivalent λ-term to tis $\lambda y.(\lambda x.xx)(\lambda x.xx)$, which is a *weak* head normal form without a (head) normal

[6] In other intersection type systems (*e.g.* [11]) $\phi \to \omega$ is a permissible type, but is equivalent to ω (that is $\omega \leq (\phi \to \omega) \leq \omega$) and so semantics based on these type systems identify terms of type $\phi \to \omega$ with unsolvable terms.

form. The 'functional' view is that such terms are observationally indistinguishable from unsolvable terms. When encoded in $\text{FJ}^{\mathfrak{C}}$ 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 $\text{FJ}^{\mathfrak{C}}$ does not have such a property. Consider the following program: `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 $\{\mathtt{C}, \langle \mathtt{m}:() \to \mathtt{C}\rangle, \langle \mathtt{m}:() \to \langle \mathtt{m}:() \to \mathtt{C}\rangle\rangle, \ldots\}$. 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 $\langle \mathtt{m}:() \to \langle \mathtt{m}:() \to \ldots\rangle\rangle$, which can be finitely represented by the recursive type $\mu X.\langle \mathtt{m}:() \to X\rangle$. 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 suprising 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 futher 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

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.
2. S. van Bakel. Intersection Type Assignment Systems. *TCS*, 151(2):385–435, 1995.
3. S. van Bakel. Cut-Elimination in the Strict Intersection Type Assignment System is Strongly Normalising. *NDJFL*, 45(1):35–63, 2004.
4. S. van Bakel. Completeness and Partial Soundness Results for Intersection & Union Typing for $\overline{\lambda}\mu\tilde{\mu}$. *APAL*, 161:1400–1430, 2010.
5. S. van Bakel and U. de'Liguoro. Logical equivalence for subtyping object and recursive types. *ToCS*, 42(3):306–348, 2008.
6. S. van Bakel and M. Fernández. Normalisation Results for Typeable Rewrite Systems. *IaC*, 2(133):73–116, 1997.
7. S. van Bakel and M. Fernández. Normalisation, Approximation, and Semantics for Combinator Systems. *TCS*, 290:975–1019, 2003.
8. S. van Bakel and R. Rowe. Semantic Predicate Types for Class-based Object Oriented Programming. In *FTfJP'09*, 2009.
9. A. Banerjee and T.P. Jensen. Modular Control-Flow Analysis with Rank 2 Intersection Types. *MSCS*, 13(1):87–124, 2003.
10. H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, 1984.
11. H. Barendregt, M. Coppo, and M Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *JSL*, 48(4):931–940, 1983.
12. L. Cardelli and J.C. Mitchell. Operations on Records. *MSCS*, 1(1):3–48, 1991.
13. L. Cardelli. A Semantics of Multiple Inheritance. *IaC*, 76(2/3):138–164, 1988.
14. M. Coppo and M Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-Calculus. *NDJFL*, 21(4):685–693, 1980.
15. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.
16. H.B. Curry. Grundlagen der Kombinatorischen Logik. *AJM*, 52:509–536, 789–834, 1930.
17. F. Damiani and F. Prost. Detecting and Removing Dead-Code using Rank 2 Intersection. In *TYPES'96*, LNCS 1512, pp 66–87, 1998.
18. K. Fisher, F. Honsell, and J.C. Mitchell. A lambda Calculus of Objects and Method Specialization. *NJ*, 1(1):3–37, 1994.
19. A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
20. T.P. Jensen. Types in Program Analysis. In LNCS 2566, pp 204–222. Springer, 2002.
21. J.C. Mitchell. Type Systems for Programming Languages. In *Handbook of TCS*, volume B, chapter 8, pages 415–431, 1990.
22. Hiroshi Nakano. A Modality for Recursion. In *LICS*, pages 255–266, 2000.
23. M. Parigot. An algorithmic interpretation of classical natural deduction. In *LPAR'92*, LNCS 624, pp 190–201, 1992.
24. G.D. Plotkin. The origins of structural operational semantics. *JLAP*, 60-61:3–15, 2004.
25. D. Scott. Domains for Denotational Semantics. In *ICALP'82*, LNCS 140, pp 577–613, 1981.
26. W.W. Tait. Intensional interpretation of functionals of finite type I. *JSL*, 32(2):198–223, 1967.
27. C.P. Wadsworth. The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus. *SIAM J. Comput*, 5:488–521, 1976.