Semantic Predicate Types and Approximation for Class-based Object Oriented Programming

Steffen van Bakel and Reuben N. S. Rowe

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2AZ, U.K.
{svb,rnr07}@doc.ic.ac.uk

Abstract. We define a small functional calculus that expresses class-based object oriented features and is modelled on the similar calculi of Featherweight Java [34] and Middleweight Java [15], which are ultimately based upon the Java programming language. We define a *predicate* system, similar to the one defined in [10], and show subject reduction and expansion, and argue that program analysis systems can be built on top of this system. Generalising the concept of approximant from the Lambda Calculus, we show that all expressions that we can assign a predicate to have an approximant that satisfies the same predicate. From this, a characterisation of (head-)normalisation follows.

1 Introduction

The *functional* programming paradigm, exemplified by languages such as ML [46], was developed around a pre-existing theoretical framework based upon the Lambda Calculus (λ-calculus) [12]. The *object oriented* programming paradigm, although developed in parallel with functional languages, did not arise from a similar such framework: the language Simula [23] was born out of a more practical need for implementing simulation software. Object orientation was then popularised in the 80s by the language Smalltalk [31], and later C++ [57]. This paradigm has now become hugely popular thanks to the languages Java [32] and C# [44].

It was only after the introduction of object oriented programming that attempts were made to place it on the same theoretical foundations as functional programming. The first were based around extending the λ-calculus and representing objects as records [16, 48, 17, 29]. The seminal work of Abadi and Cardelli [1] constitutes perhaps the most comprehensive formal treatment of object orientation, and introduces the Varsigma object calculus (ς-calculus). This calculus is a highly abstract view of the object oriented programming paradigm, and describes many features found in a multiplicity of programming languages. Despite this generality, the ς-calculus was formulated using a particular view of the object oriented methodology – the *object-based* variety as opposed to the *class-based* variety. For this reason, work was carried out to develop similar formal models describing class-based languages. Notable efforts are Featherweight Java [34] and its successor Middleweight Java [15].

An integral aspect of the theory of programming languages is *type theory* which allows for static analysis via abstract reasoning about programs, so that certain guarantees

can given about their behaviour. Type theory arose side-by-side with the formal models, one of the earliest being Curry's system for the λ-calculus [22]. Type theory easily found acceptance within the world of programming, not only through Milner's claim "*typed programs cannot go wrong*"[1], but also because static, compile time type analysis allows for efficient code generation, and the generation of efficient code. The quest for expressive type systems is still ongoing; for example, types with quantifiers [30, 52] were investigated in the early nineties [47, 50, 51, 18], and the *intersection type discipline* (ITD), as first developed in the early 1980s [19, 20, 13, 4] are two good examples of, in principle, undecidable systems that have found practical application.

ITD generalises Curry's system by allowing more than one type for free and bound variables, grouped in *intersections* via the type constructor $\cap$. This (for free variables) is a feature also of ML [45, 24], where, in the `let`-rule, a term-variable x can have a generic type which can get instantiated into several types, effectively typing x with all those types. In fact, the ML system has similarities with Rank 2 intersection type assignment [40, 3]; the key idea for this system is to restrict the set of types to those of the shape $((\sigma_1 \cap \ldots \cap \sigma_n) \to \tau)$, where the σ_i are types that do not contain intersections. This kind of types later were used outside the context of the λ-calculus in, e.g., [5, 25, 6, 59]. Since then, many decidable restrictions of various ranks have been defined by A. Kfoury and J. Wells [38, 37]. However, in contrast to ML, ITD also allows more than one type for *bound* variables, effectively opening the possibility to accept arguments of more than one type (note that this is not the same as accepting an argument of *polymorphic* type, which would correspond to System F [30]: there are terms typeable in ITD that are not typeable in System F). This slight generalisation causes a great change in complexity; in fact, now all terms having a (head-)normal form can be characterised by their assignable types (see also Section 7), a property that immediately shows that type assignment is undecidable, since it would solve the halting problem. Also, by introducing this extension a system is obtained that is closed under β-equality: if $B \vdash M : \sigma$ and $M =_\beta N$, then $B \vdash N : \sigma$ (see also Section 4.1).

So intersection systems are too powerful. But they satisfy a number of strong properties, that are preserved even when considering decidable restrictions. For example, soundness (subject reduction) will always hold, as does the fact that a term that satisfies certain criteria will terminate (has a normal form), or, with different criteria, produce output (has a head-normal form). What is normally lost when considering a restriction is the '*only if*' part of the properties. And, in fact, intersection types have found their way not only into programming languages [53–55, 39, 26, 28, 49], but have long since proved their worth within the area of *abstract interpretation* [35, 21, 14].

The strength of ITD motivated de'Liguoro [27] to apply the principles of intersection types to object oriented programming, in particular to the Varsigma Calculus. Over three papers [8–10], several systems were explored, for various variants of that calculus.

In this work, we aim to follow up on these efforts and apply the principles of intersection types, and the system of [10] specifically, to a formal model of *class-based* object oriented programming. The model that we use is based on [34, 15]. We find that we would like to use a slightly richer calculus than [34], but that the collection of fea-

[1] Here '*wrong*' is a semantic value that represents an error state, created when, for example, trying to apply a number to a number

tures in [15] is, at least for the moment, too complex for our purposes. Therefore, we define a new calculus, which we call *Lightweight Java*. Having defined the calculus, we will then prove subject reduction and expansion results.

Although the system we study here is based on the one presented in [10], and thereby firmly rooted in the area of intersection types [19, 20, 13, 4, 7], this paper requires no previous knowledge of those systems at all. In fact, the intersection type constructor is not used at all in our system, albeit present implicitly in the *join* operation. We choose this approach to highlight the strength of the system, the main idea of which is to allow more than one analysis (grouped in an object-predicate) to be used simultaneously for an expression.

As with intersection types for the λ-calculus, the types we introduce here open the way for defining semantics, building a filter model by interpreting expressions by their assignable predicates. This is a straightforward extension of the results of this paper, but is omitted because of space restrictions: we focus here on the presentation of the calculus and the predicate assignment system, and the fact that this is closed for reduction and expansion of typed terms; this then should sufficiently illustrate the semantic properties of the system.

The goal of our research is to come to a semantics-based or type-based abstract interpretation for object orientation, for which the present paper contains the first steps. While the abstract interpretation of object-oriented languages has certainly been an active topic of research, the majority of approaches taken thus far appear to have concentrated on control-flow and data-flow analysis techniques rather than type-based abstractions [43]. An exception to this is found in [33]. Another observation is that work in this area has been centred around issues of optimisation: [36] presents a *class analysis* of object-oriented programs which may be used to eliminate virtual function calls, *pointer analysis* [56] generalises class analysis and also allows for the detection of null pointer dereferencing, and other analyses [41, 42] have looked at inferring invariants for classes which can be useful in many optimisations such as the removal of checks for array bounds. What is missing from this list is termination analysis, which is what our treatment addresses. A termination analysis of Java bytecode has been done [2], however our system aims at performing such an analysis directly at the level of the object-oriented language rather than its intermediate form. We also note that while our work takes termination analysis as its starting point, it is not necessarily limited to this type of analysis.

The normal, class-based type system for our variant of Java is sound, but not expressive enough to come to in-depth analysis of programs; we therefore introduce the additional concept of predicates, which express the functional behaviour of programs, and allow their execution to be traced. We show that the standard (functional) properties hold and, moreover, put in evidence that we have a strong semantic system: we can prove an approximation result and characterise head-normalisation. The system, being semi-decidable at best, would need to be limited in expressiveness before it can be used for static analysis, but the main properties shown in this paper would hold also for such a restriction; the one to fail would be the *completeness* side of things, where we reason against the flow of execution.

2 A Class-based Calculus: Lightweight Java

In this section, we formally define the calculus that we study. It is based on aspects of both Featherweight Java (FJ) and Middleweight Java (MJ), and so to continue with the theme we have named it *Lightweight Java* (LJ). We retain the functional nature of FJ, but add some features from MJ (namely field assignment expressions and `null` objects) which do not conflict with this.

The following notational conventions will be used:

1. We use $\overline{n}$ (where n is natural number) to represent the list $1, \ldots, n$.
2. A sequence of n elements $a_1, \ldots, a_n$ is denoted by $\overrightarrow{a_n}$. The subscript can be omitted when the exact number of elements in the sequence is not relevant. Both comma-separated and space-separated sequences may be abbreviated in this way.
3. We write $a \in \overrightarrow{a_n}$ whenever there exists some $j \in \{1, \ldots, n\}$ such that $a_j = a$. Similarly, we write $a \notin \overrightarrow{a_n}$ whenever there does *not* exist an $i \in \{1, \ldots, n\}$ such that $a_i = a$. Again, the subscript n may be omitted as implicit.
4. The empty sequence is denoted by ϵ.
5. If F is a partial function defined over n arguments, then $F(\overrightarrow{\text{arg}})\downarrow$ ($F(\overrightarrow{\text{arg}})\uparrow$) denotes that F is (not) defined on the arguments $\overrightarrow{\text{arg}}$.

Definition 1 (Identifiers). We define the following sets of identifiers:

1. The set CLASS-NAME of *class names* which includes the distinguished element `Object`, and is ranged over by C, D.
2. The set FIELD-ID of field identifiers, ranged over by f.
3. The set METHOD-NAME of method names, ranged over by m.
4. l ranges over the union of the set of field identifiers and method names, which we call the set of *class member labels*.
5. The set VARIABLE of variables, which includes the special variable `this`, ranged over by x.

Note that C, D, f, m and x are *meta-variables* which range over sets. However, when these letters are written in a fixed-width font (`C`, `D`, `f`, `m`, `x`), this indicates that we are referring to actual members of the sets, and not meta-variables.

As in FJ and MJ (and indeed full Java [32]), types are embedded within the syntax of the language itself, allowing the programmer to explicitly specify the type of each field and method. Thus, in order to define the syntax of our calculus, we must first define types. Notice that, for simplicity, we do not assume the presence of basic types, like INT or CHAR, nor constants; these can easily be added without affecting our results.

Definition 2 (Class and method types). 1. The types that can be assigned to LJ expressions are called *class* types, corresponding to the intuition that each expression results in an object which is an instance of some class. The set of class types is identical to the set of class names and, as such, we will also use C and D to range over these types. As will be seen in the syntax definition below, we also use class types as annotations in field definitions since fields are used to store object values.

2. The *method types* are defined by the following grammar:

$$\mu \ ::= \ \mathrm{C}_1, \ldots, \mathrm{C}_n \to \mathrm{D}$$

Thus, each method takes a sequence of arguments of types $\mathrm{C}_1, \ldots, \mathrm{C}_n$, and returns a result of type D.

Definition 3 (Program syntax). 1. We define a set of *expressions* ranged over by e. We also define a subset of expressions ranged over by o, which is the set of *objects*:

$$\begin{aligned} \mathsf{e} &::= x \mid (\mathrm{C})\,\mathtt{null} \mid \mathsf{e}.f \mid \mathsf{e}.f = \mathsf{e}' \mid \mathsf{e}.m(\vec{\mathsf{e}}) \mid \mathtt{new}\ \mathrm{C}(\vec{\mathsf{e}}) \\ \mathsf{o} &::= x \mid (\mathrm{C})\,\mathtt{null} \mid \mathtt{new}\ \mathrm{C}(\vec{\mathsf{o}}) \end{aligned}$$

Notice that the expression $\mathtt{new}\ \mathrm{C}(\vec{\mathsf{e}})$ is an object only when all e_i are.

2. *Classes* are defined by the following grammar:

$$\begin{aligned} \mathsf{fd} &::= \mathrm{C}\ f \\ \mathsf{md} &::= \mathrm{D}\ m(\overrightarrow{\mathrm{C}\ x})\ \{\, \mathsf{e} \,\} \\ \mathsf{cd} &::= \mathtt{class}\ \mathrm{C}\ \mathtt{extends}\ \mathrm{C}'\ \{\, \vec{\mathsf{fd}}\ \vec{\mathsf{md}} \,\} \quad (\mathrm{C} \neq \mathtt{Object}) \end{aligned}$$

3. LJ Programs consist of an *execution context* (which is a sequence of class definitions), and an expression that is evaluated when the program is run:

$$\begin{aligned} \mathcal{EC} &::= \vec{\mathsf{cd}} \\ P &::= (\mathcal{EC}, \mathsf{e}) \end{aligned}$$

We now discuss these various syntactic elements. Expressions may create objects that conform to a specified class template indicated by C, using the `new` keyword. They may also invoke methods, and retrieve and assign field values to objects or parameters. Expressions may also refer to variables (method parameters) and the null value.

Classes contain a list of fields and a list of methods, the types of which must be declared. Methods may take multiple arguments, and method bodies consist of a single expression. Classes may also inherit from one another, meaning that they share field and method definitions. There is a lack of symmetry between the structure of field definitions and method definitions: in a field definition, the type of the field precedes the field identifier, whereas the sequence of parameter types comprising the method type is mixed in with the sequence of formal parameters. However this is an intentional decision, motivated by the desire to have the calculus conform to its predecessors (LJ and MJ), as well as its namesake language.

Note that fields are more than just parameter-less methods: we allow the values returned by fields to be updated during program execution, whereas the behaviour of methods is fixed, once and for all, when a class is defined. However, the syntax of Definition 3 does not disallow the *redeclaration* of any methods and fields that have be defined in a superclass. We will specify that for an execution context to be well-formed (and so in turn for expressions to be typeable), classes must not redeclare fields in this way. We will also specify that if methods are redeclared, then the method type must match that of the superclass, although the names of the formal parameters may differ.

We will not place any restrictions on the *body* of the redeclared method, thus allowing a limited type-restricted form of method *override*.

Unlike in FJ and MJ, and indeed full Java itself, we do not include object constructor methods in the language definition. Full Java allows objects to be created in multiple ways, thereby requiring the ability to define multiple constructor methods. Both FJ and MJ enforce a single constructor method for each class to which initial values for all fields must be passed as parameters. There is no loss of generality in this approach and so we adopt it for LJ. However, in FJ and MJ, constructor methods are made explicit and must appear in the method definition list of each class. This constructor method is distinguished from the other methods through the use of separate typing rules. We feel that this is an unnecessary complication of the language syntax, and so make the constructor method *implicit* by requiring in the type rule for the `new` keyword that the types of the expressions appearing in the object creation construct match the types for the sequence of fields defined by the class of the object being created.

Furthermore, we have chosen to omit *cast expressions* from our language. It appears that casts were included in FJ in order to support the compilation of FGJ programs to FJ [34, §3]. Since that is not an objective of the current work, and the presence of downcasts makes the system unsound (in the sense that well-typed expressions can get stuck), they are omitted. Upcasts are replaced by subsumption rules in the type system. At this stage, we point out a slight departure from the usual Java syntax in our definition of `null` object values. In LJ, each `null` value is annotated with a class type. Note that this is *not* casting, since we do not define a special *null* type, but is simply a type declaration similar in spirit to those of field and method definitions.

Definition 4 (Syntax look-up functions). We define the following look-up functions to retrieve the various syntactic elements of a program:

1. The following three functions retrieve the names of a class, a method and a field from their respective definitions:

$$\begin{array}{ll} \mathrm{CN}\,(\texttt{class C extends C}'\,\{\,\overrightarrow{\mathsf{fd}}\;\overrightarrow{\mathsf{md}}\,\}) & \triangleq\ \mathrm{C} \\ \mathrm{MN}\,(\mathrm{D}\ m(\overrightarrow{\mathrm{C}\ x})) & \triangleq\ m \\ \mathrm{FN}\,(\mathrm{C}\ f) & \triangleq\ f \end{array}$$

2. The *class table*, $\mathcal{CT}$, is a partial map from execution contexts and class names to class definitions:

$$\mathcal{CT}\,(\mathcal{EC}, \mathrm{C}) \ \triangleq\ \begin{cases} \mathsf{cd} & \mathrm{CN}\,(\mathsf{cd}) = \mathrm{C}\ \&\ \mathsf{cd} \in \mathcal{EC} \\ \text{Undefined} & \text{otherwise} \end{cases}$$

   We emphasize that the class table is to be undefined on the special class `Object`, since this class should only serve as the root of the class hierarchy and contain no fields and methods.
3. The *SuperCl* function is a partial map from execution contexts and class names to class names, returning the direct superclass of a given class within the given context.

4. The list of fields belonging to a class C in an execution context $\mathcal{EC}$ is given by the following function:

$$\mathcal{F}(\mathcal{EC}, \mathrm{C}) \triangleq \begin{cases} \mathcal{F}(\mathcal{EC}, \mathrm{C}') \cdot \overrightarrow{f_n} & \mathcal{CT}(\mathcal{EC}, \mathrm{C}) = \texttt{class C extends C}' \{ \overrightarrow{\mathsf{fd}_n}\ \overrightarrow{\mathsf{md}} \} \\ & \quad \& \forall i \in \overline{n} \, [\, f_i = \mathrm{FN}(\mathsf{fd}_i) \,] \\ \epsilon & \text{otherwise} \end{cases}$$

   Thus, the sequence returned by this function contains not only the fields declared in the specified class, but all the fields that it inherits from its superclasses.
5. The function $\mathcal{M}$ returns a set of method names for a given class, corresponding to the methods declared and inherited by that class. It is defined as follows:

$$\mathcal{M}(\mathcal{EC}, \mathrm{C}) \triangleq \begin{cases} \mathcal{M}(\mathcal{EC}, \mathrm{C}') \cup \{ \mathrm{MN}(\mathsf{md}) \mid \mathsf{md} \in \overrightarrow{\mathsf{md}} \} & \\ \qquad\qquad \mathcal{CT}(\mathcal{EC}, \mathrm{C}) = \texttt{class C extends C}' \{ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}} \} & \\ \emptyset & \text{otherwise} \end{cases}$$

   Note that we have defined this function to return a set rather than a list since, unlike fields, the order of methods is *not* important.
6. The function $\mathcal{M}b$, when given an execution context $\mathcal{EC}$, class name C and method name m, returns a tuple $(\vec{x}, \mathsf{e})$, consisting of a sequence of the method's formal parameters and the method body:

$$\mathcal{M}b(\mathcal{EC}, \mathrm{C}, m) \triangleq \begin{cases} (\vec{x}, \mathsf{e}) & \mathcal{CT}(\mathcal{EC}, \mathrm{C}) = \texttt{class C extends C}' \{ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}} \} \\ & \quad \& \ \mathrm{C}_0\ m(\overrightarrow{\mathrm{C}\ x}) \{ \mathsf{e} \} \in \overrightarrow{\mathsf{md}} \\ \mathcal{M}b(\mathcal{EC}, \mathrm{C}', m) & \mathcal{CT}(\mathcal{EC}, \mathrm{C}) = \texttt{class C extends C}' \{ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}} \} \\ & \quad \& \ \mathrm{C}_0\ m(\overrightarrow{\mathrm{C}\ x}) \{ \mathsf{e} \} \notin \overrightarrow{\mathsf{md}} \\ \text{Undefined} & \mathcal{CT}(\mathcal{EC}, \mathrm{C})\uparrow \end{cases}$$

We now define look-up functions which allow us to extract the type information that is defined in a given class:

Definition 5 (Member type lookup). The *field table* $\mathcal{FT}$ and *method table* $\mathcal{MT}$ are functions which return type information about the elements of a given class within an execution context. These functions allow us to retrieve the types of any given field f or method m declared in a particular class C in the context $\mathcal{EC}$:

$$\mathcal{FT}(\mathcal{EC}, \mathrm{C}, f) = \begin{cases} \mathrm{D} & \mathcal{CT}(\mathcal{EC}, \mathrm{C}) = \texttt{class C extends C}' \{ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}} \} \\ & \quad \& \ \mathrm{D}\ f \in \overrightarrow{\mathsf{fd}} \\ \mathcal{FT}(\mathcal{EC}, \mathrm{C}', f) & \mathcal{CT}(\mathcal{EC}, \mathrm{C}) = \texttt{class C extends C}' \{ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}} \} \\ & \quad \& \ \mathrm{D}\ f \notin \overrightarrow{\mathsf{fd}} \\ \text{Undefined} & \mathcal{CT}(\mathcal{EC}, \mathrm{C})\uparrow \end{cases}$$

$\mathcal{MT}$ is defined by:

$$\mathcal{MT}(\mathcal{EC}, \mathrm{C}, m) = \begin{cases} \overrightarrow{\mathrm{C}_n} \to \mathrm{D} & \mathcal{CT}(\mathcal{EC}, \mathrm{C}) = \texttt{class C extends C}' \{ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}} \} \\ & \quad \& \ \mathrm{D}\ m(\overrightarrow{\mathrm{C}\ x}) \{ \mathsf{e} \} \in \overrightarrow{\mathsf{md}} \\ \mathcal{MT}(\mathcal{EC}, \mathrm{C}', m) & \mathcal{CT}(\mathcal{EC}, \mathrm{C}) = \texttt{class C extends C}' \{ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}} \} \\ & \quad \& \ \mathrm{D}\ m(\overrightarrow{\mathrm{C}\ x}) \{ \mathsf{e} \} \notin \overrightarrow{\mathsf{md}} \\ \text{Undefined} & \mathcal{CT}(\mathcal{EC}, \mathrm{C})\uparrow \end{cases}$$

Notice that this look-up function is undefined for the special class `Object`, which does not have any fields or methods.

We say that C *extends* C' *in the context* $\mathcal{EC}$ when $\mathcal{CT}(\mathcal{EC},\mathrm{C}) = \texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{C}'\ \{\ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}}\ \}$. We say that C is a *subclass* of C' in the context $\mathcal{EC}$ when C extends C', or there is a non-empty sequence of classes $\mathrm{C}_1,\ldots,\mathrm{C}_n$ in $\mathcal{EC}$ such that C extends C_1, each C_i extends C_{i+1} for all $i \in \overline{n-1}$, and C_n extends C'. We formalise this notion by:

Definition 6 (Subtypes). For a context $\mathcal{EC}$, the *subtype relation* is defined as the smallest transitive relation satisfying: $\mathit{SuperCl}(\mathcal{EC},\mathrm{C}) = \mathrm{C}' \ \Rightarrow\ \mathrm{C} <:_{\mathcal{EC}} \mathrm{C}'$

Finally, we define *validity* of a type. We say that a type is *valid* with respect to an execution context $\mathcal{EC}$, written $\vdash_{\mathcal{EC}} \mathrm{C}$, when the corresponding class is defined in the context.

Definition 7 (Valid types). We define type validity through the following judgements:

$$\frac{}{\vdash_{\mathcal{EC}} \texttt{Object}} \qquad \frac{}{\vdash_{\mathcal{EC}} \mathrm{C}}\ (\mathcal{CT}(\mathcal{EC},\mathrm{C})\downarrow) \qquad \frac{\vdash_{\mathcal{EC}} \mathrm{C}_1 \quad \ldots \quad \vdash_{\mathcal{EC}} \mathrm{C}_n \quad \vdash_{\mathcal{EC}} \mathrm{D}}{\vdash_{\mathcal{EC}} \overrightarrow{\mathrm{C}} \to \mathrm{D}}$$

Well formedness of a context is itself a necessary condition for the correct reduction (execution) of LJ programs.

Definition 8 (Well-formed context). An execution context $\mathcal{EC} = \overrightarrow{\mathsf{cd}_n}$ is well formed ($\vdash \mathcal{EC}$) if, and only if, it satisfies the following conditions:

1. There are no duplicate class definitions: $\forall\, i,j \in \overline{n}\ [\ i \neq j \Rightarrow \mathrm{CN}\,(\mathsf{cd}_i) \neq \mathrm{CN}\,(\mathsf{cd}_j)\ ]$.
2. The class hierarchy is *acyclic*.
3. All fields defined in a particular branch of the class hierarchy are uniquely named.
4. There are no duplicate method declarations within a given class, and the types of any overridden methods must match.
5. The special variable `this` must not appear as a parameter in any method definition.
6. All types declared for fields and methods must be valid types with respect to the execution context, as must all classes that are inherited from.

Property 9 (Type consistency). We note that well-formed execution contexts exhibit the following consistency properties:

1. The types of inherited fields in a subclass are consistent with the type of the field in the superclass:

$$\begin{array}{rcl} \vdash \mathcal{EC}\ \&\ \mathcal{FT}(\mathcal{EC},\mathrm{C},f) & = & \mathrm{C}'\ \& \\ \mathcal{FT}(\mathcal{EC},\mathrm{D},f) & = & \mathrm{D}'\ \& \\ \mathrm{C} <:_{\mathcal{EC}} \mathrm{D} & \Rightarrow & \mathrm{C}' = \mathrm{D}' \end{array}$$

2. The types of inherited and overridden methods are consistent with the type of the method in the superclass:

$$\begin{array}{rcll} \vdash \mathcal{EC}\ \&\ \mathcal{MT}(\mathcal{EC},\mathrm{C},m) & = & \overrightarrow{\mathrm{C}_{n_1}} \to \mathrm{C}_0\ \& & \\ \mathcal{MT}(\mathcal{EC},\mathrm{D},m) & = & \overrightarrow{\mathrm{D}_{n_2}} \to \mathrm{D}_0\ \& & \\ \mathrm{C} <:_{\mathcal{EC}} \mathrm{D} & & \Rightarrow & \mathrm{C}_0 = \mathrm{D}_0\ \&\ n_1 = n_2\ \& \\ & & & \forall\, i \in \overline{n}\ [\ \mathrm{C}_i = \mathrm{D}_{\ i}] \end{array}$$

We will now define a reduction relation, $\rightarrow$, on LJ programs.

Definition 10 (Reduction). The one-step reduction relation is defined by the following rules:

(R-FLD): $(\mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}})).f_i \rightarrow_{\mathcal{EC}} \mathsf{e}_i$, if $\mathcal{F}(\mathcal{EC}, \mathrm{C}) = \overrightarrow{f_n}$ & $i \in \overline{n}$;
(R-ASS): $(\mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}})).f_j = \mathsf{e}'_j \rightarrow_{\mathcal{EC}} \mathsf{new}\ \mathrm{C}(\mathsf{e}_1, \ldots, \mathsf{e}'_j, \ldots, \mathsf{e}_n)$, if $\mathcal{F}(\mathcal{EC}, \mathrm{C}) = \overrightarrow{f_n}$, $j \in \overline{n}$,
(R-INVK): $(\mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}})).m(\overrightarrow{\mathsf{e}'}) \rightarrow_{\mathcal{EC}} \mathsf{e}[\overrightarrow{\mathsf{e}'/x}, \mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}})/\mathtt{this}]$, if $\mathcal{M}b(\mathcal{EC}, \mathrm{C}, m) = (\vec{x}, \mathsf{e})$;
(RC-FLD): If $\mathsf{e} \rightarrow_{\mathcal{EC}} \mathsf{e}'$, then $\mathsf{e}.f \rightarrow_{\mathcal{EC}} \mathsf{e}'.f$;
(RC-ASS_1): If $\mathsf{e} \rightarrow_{\mathcal{EC}} \mathsf{e}''$, then $(\mathsf{e}.f = \mathsf{e}') \rightarrow_{\mathcal{EC}} (\mathsf{e}''.f = \mathsf{e}')$;
(RC-ASS_2): If $\mathsf{e}' \rightarrow_{\mathcal{EC}} \mathsf{e}''$, then $(\mathsf{e}.f = \mathsf{e}') \rightarrow_{\mathcal{EC}} (\mathsf{e}.f = \mathsf{e}'')$;
(RC-INVK_1): If $\mathsf{e}_0 \rightarrow_{\mathcal{EC}} \mathsf{e}'_0$, then $\mathsf{e}_0.m(\vec{\mathsf{e}}) \rightarrow_{\mathcal{EC}} \mathsf{e}'_0.m(\vec{\mathsf{e}})$;
(RC-INVK_2): If $\mathsf{e}_j \rightarrow_{\mathcal{EC}} \mathsf{e}'_j$, then $\mathsf{e}_0.m(\vec{\mathsf{e}}) \rightarrow_{\mathcal{EC}} \mathsf{e}_0.m(\mathsf{e}_1, \ldots, \mathsf{e}'_j, \ldots, \mathsf{e}_n)$, if $j \in \overline{n}$.
(RC-NEW): If $\mathsf{e}_j \rightarrow_{\mathcal{EC}} \mathsf{e}'_j$, then $\mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}}) \rightarrow_{\mathcal{EC}} \mathsf{new}\ \mathrm{C}(\mathsf{e}_1, \ldots, \mathsf{e}'_j, \ldots, \mathsf{e}_n)$, if $j \in \overline{n}$.

If $\mathsf{e} \rightarrow_{\mathcal{EC}} \mathsf{e}'$, we call e the *redex*, and e' the *reduct*. We also say that e' *expands* to e. We denote the transitive closure of $\rightarrow$ by $\rightarrow^*$, and thus write $\mathsf{e} \rightarrow^*_{\mathcal{EC}} \mathsf{e}'$ if there exists a (possibly empty) sequence $\vec{\mathsf{e}}$ such that $\mathsf{e} \rightarrow_{\mathcal{EC}} \mathsf{e}_1 \rightarrow_{\mathcal{EC}} \ldots \rightarrow_{\mathcal{EC}} \mathsf{e}_n \rightarrow_{\mathcal{EC}} \mathsf{e}'$.

We should point out that we have a notion of 'free' reduction, in that there might be more than one reducible expression (redex) in a program, and that the reduction rules above do *not* define a strategy, i.e. no redex is preferred over others. This is a generalisation of the normal situation, since traditionally a notion of "lazy" reduction is used, notably omitting the rules (RC-ASS_2), (RC-INVK_2) and (RC-NEW), so we have more possible reduction paths. However, our main approximation result is shown for this free reduction, so is stronger than one we could have obtained for lazy reduction.

This notion of reduction is *confluent*, which can be shown by the standard 'colouring' argument. Of course confluence can easily be achieved by restricting reduction to lazy reduction.

3 The Type System

We define two type assignment systems for LJ expressions. The first directly corresponds to the type systems found in [34] and [15], which in turn are modelled on the full Java type system [32]. We call this the *class* type system. The second system is the *predicate* type system. For readability, when we refer to the type system from now on, we will mean the *class* type system, and the *predicate system* will refer to the predicate type system. When we refer to *types*, we will mean class types (as used in the class type system), and similarly *predicates* will refer to predicate types.

The predicate system takes after the system of the same name in [10], with predicates comprising sequences of statements, each of which describes a single behaviour exhibited by the expression to which it is assigned.

We start our treatment of type assignment by defining type environments.

$$[\text{T-NULL}]: \frac{\mathcal{EC} \vdash \Gamma \quad \vdash_{\mathcal{EC}} \mathrm{C}}{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} (\mathrm{C})\ \mathtt{null}{:}\mathrm{C}} \qquad [\text{T-VAR}]: \frac{\mathcal{EC} \vdash \Gamma}{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} x{:}\mathrm{C}}\ (x{:}\mathrm{C} \in \Gamma)$$

$$[\text{T-FLD}]: \frac{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{D}}{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}.f{:}\mathrm{C}}\ (\mathcal{FT}(\mathcal{EC}, \mathrm{D}, f) = \mathrm{C}) \qquad [\text{T-SUB}]: \frac{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}'}{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}}\ (\mathrm{C}' <:_{\mathcal{EC}} \mathrm{C})$$

$$[\text{T-ASS}]: \frac{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{D} \quad \Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}'{:}\mathrm{C}}{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}.f = \mathsf{e}'{:}\mathrm{D}}\ (\mathcal{FT}(\mathcal{EC}, \mathrm{D}, f) = \mathrm{C})$$

$$[\text{T-INVK}]: \frac{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} \quad \Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}_i{:}\mathrm{C}_i\ (\forall i \in \overline{n})}{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}.m(\vec{\mathsf{e}}){:}\mathrm{D}}\ (\mathcal{MT}(\mathcal{EC}, \mathrm{C}, m) = \vec{\mathrm{C}} \to \mathrm{D})$$

$$[\text{T-NEW}]: \frac{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}_i{:}\mathrm{C}_i \quad (\forall i \in \overline{n})}{\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{new}\ \mathrm{C}(\vec{\mathsf{e}}){:}\mathrm{C}}\ (\mathcal{F}(\mathcal{EC}, \mathrm{C}) = \vec{f}\ \&\ \mathcal{FT}(\mathcal{EC}, \mathrm{C}, f_i) = \mathrm{C}_i\ (\forall i \in \overline{n}))$$

Fig. 1. Type assignment rules

Definition 11 (Type environments). 1. A *type statement* is of the form $\mathsf{e}{:}\mathrm{C}$, where e is an LJ expression, and C is a class type. The expression e is called the *subject* of the statement, and the type C is called the *conclusion* of the statement.
2. A *type environment*, Γ, is a set of type statements with term variables as subjects.
3. We use the notation $\Gamma, x{:}\mathrm{C}$ to represent $\Gamma \cup \{x{:}\mathrm{C}\}$. Similarly, we write Γ, Γ' to represent $\Gamma \cup \Gamma'$.

We write $x \in \Gamma$ whenever there is a C such that $x{:}\mathrm{C} \in \Gamma$.

Note that we do not require the term variables in a type environment to be distinct. This is only the case for *well formed* environments, defined below.

Definition 12 (Well-formed type environments). We say that a type environment Γ is *well formed* with respect to some execution context $\mathcal{EC}$, when the execution context is itself well formed and the statements in Γ all have distinct variables as subjects, and the conclusion of each statement is a valid class type with respect to $\mathcal{EC}$. This notion is formalised through the following judgements:

$$\frac{\vdash \mathcal{EC}}{\mathcal{EC} \vdash \emptyset} \qquad \frac{\mathcal{EC} \vdash \Gamma \quad \vdash_{\mathcal{EC}} \mathrm{C}}{\mathcal{EC} \vdash \Gamma, x{:}\mathrm{C}}\ (\neg\exists \mathrm{D}\ [\ x{:}\mathrm{D} \in \Gamma\ ])$$

Definition 13 (Type assignment). Type assignment is a ternary relation between execution contexts, type environments and type statements, written as $\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}$. We say that the type C can be *assigned* to the expression e in the context $\mathcal{EC}$ using the type environment Γ. The relation $\vdash^{\mathrm{T}}$ is defined using the natural deduction system of Figure 1.

Definition 14 (Type consistent execution contexts). We say that an execution context is *type consistent* ($\vdash^{\mathrm{T}}_{\mathcal{EC}} \diamond$) when the bodies of all methods defined in the context can be assigned their declared return type.

$$\vdash^{\mathrm{T}}_{\mathcal{EC}} \diamond \ \Leftrightarrow\ \vdash \mathcal{EC}\ \&\ \forall \mathrm{C}\ [\ \mathcal{CT}(\mathcal{EC}, \mathrm{C})\downarrow \Rightarrow \forall m\ [\ \mathcal{MT}(\mathcal{EC}, \mathrm{C}, m) = \overrightarrow{\mathrm{D}} \to \mathrm{D}_0\ \&\ \mathcal{M}b\ (\mathcal{EC}, \mathrm{C}, m) = (\vec{x}, \mathsf{e}_0) \Rightarrow \{\overrightarrow{x{:}\mathrm{D}}, \mathtt{this}{:}\mathrm{C}\} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}_0{:}\mathrm{D}_0\ ]\ ]$$

4 The Predicate System

We now define an extension to the class type system: the predicate type system. We will see that the predicate system types exactly the same set of terms as the class type system. This result is shown in Theorem 24.

Definition 15 (Predicates). The sets of predicates are inductively as follows:

$$\begin{array}{lrcll} \textit{predicates}: & \phi, \psi & ::= & \top \mid \sigma & \\ \textit{object predicates}: & \sigma & ::= & \langle l_1{:}\tau_1, \ \ldots \ , l_n{:}\tau_n \rangle & (n \geq 0) \\ \textit{member predicates}: & \tau, \rho & ::= & \phi \mid \psi :: \phi_1, \ \ldots \ , \phi_n \rightarrow \sigma & (n \geq 0) \end{array}$$

We abbreviate $\langle l_1{:}\tau_1, \ \ldots \ , l_n{:}\tau_n \rangle$, by writing $\langle l_i : \tau_i \ ^{i \in \underline{n}} \rangle$, and we call the object predicate consisting of the empty sequence, $\langle \epsilon \rangle$, the *empty* predicate. We call $\top$ a *trivial* predicate; all other predicates are, correspondingly, *non-trivial*.

The aim in defining predicates in this way is that they should describe the behaviour of an object in terms of consecutive method or field calls. The predicate member statements that comprise an object predicate each indicate that the object to which it is assigned behaves in a particular way. The class member label in each statement denotes either a field or method belonging to the object, and its associated member predicate then describes the result of accessing the field or invoking the method. In the case of a method, the member predicate $\psi :: \phi_1, \ \ldots \ , \phi_n \rightarrow \sigma$ also indicates the *required* behaviour of the arguments ($\phi_1, \ \ldots \ , \phi_n$, as well as the receiver (ψ). The universal predicate $\top$ is intended to indicate non-terminating behaviour or a discarded computation, as will be discussed in §4.2.

Definition 16 (Subpredicates). 1. The relation $\trianglelefteq$ is defined as the least pre-order on predicates such that:

$$\begin{array}{rcll} \langle \epsilon \rangle & \trianglelefteq & \top & \\ \langle l_i : \tau_i \ ^{i \in \underline{n}} \rangle & \trianglelefteq & \langle l_j{:}\tau_j \rangle & \forall j \in \overline{n} \\ \phi \trianglelefteq \langle l_j{:}\tau_j \rangle \text{ for all } j \in \overline{n} & \Rightarrow & \phi \trianglelefteq \langle l_i : \tau_i \ ^{i \in \underline{n}} \rangle & n \geq 0 \\ \phi \trianglelefteq \psi & \Rightarrow & \langle l{:}\phi \rangle \trianglelefteq \langle l{:}\psi \rangle & \end{array}$$

2. The equivalence relation $\phi \sim \psi$ is defined over member types by:

$$\phi \sim \psi \ \Leftrightarrow \ \phi \trianglelefteq \psi \trianglelefteq \phi$$

This definition captures the intuition that object predicates should be equivalent up to reordering of predicate member statements. Thus

$$\langle l_i{:}\tau_i \ ^{i \in \overline{n}} \rangle \sim \langle l'_i{:}\tau'_i \ ^{i \in \overline{m}} \rangle \ \Rightarrow \ n = m \ \& \ \forall i \in \overline{m} \ \exists j \in \overline{n} \ [\ l_j = l'_i \ \& \ \tau_j = \tau'_i \ ]$$

Definition 17 (Predicate join). 1. The *join* of two predicates, $\phi_1 \sqcup \phi_2$, is defined as follows:

$$\begin{array}{c} \phi \sqcup \top \ = \ \phi \qquad \top \sqcup \phi \ = \ \phi \qquad \phi \sqcup \phi \ = \ \phi \\ \langle l_i : \tau_i \ ^{i \in \underline{n}} \rangle \sqcup \langle l'_i : \tau'_i \ ^{i \in \underline{m}} \rangle \ = \ \langle l''_i{:}\tau''_i \ ^{i \in \overline{k}} \rangle \text{ where for each } i \in \overline{k} \end{array}$$

- $\tau_i'' = \tau$ if $l_i'':\tau \in \langle l_i : \tau_i \ ^{i \in \underline{n}} \rangle$ and $l_i'' \notin \langle l_i' : \tau_i' \ ^{i \in \underline{m}} \rangle$, and
- $\tau_i'' = \tau$ if $l_i'' \notin \langle l_i : \tau_i \ ^{i \in \underline{n}} \rangle$ and $l_i'':\tau \in \langle l_i' : \tau_i' \ ^{i \in \underline{m}} \rangle$.
- $\tau_i'' = \phi \sqcup \psi$ if $l_i'':\phi \in \langle l_i : \tau_i \ ^{i \in \underline{n}} \rangle$ and $l_i'':\psi \in \langle l_i' : \tau_i' \ ^{i \in \underline{m}} \rangle$,

2. We also overload $\sqcup$ to define the following function which generalises the notion of join to sequences of predicates:

$$\begin{aligned} \sqcup\epsilon &= \langle \epsilon \rangle \\ \sqcup\phi \cdot \overrightarrow{\phi_n} &= \phi \sqcup (\sqcup \overrightarrow{\phi_n}) \end{aligned}$$

The join operation is used in the proof of the subject expansion property for the predicate system in Theorem 27.

This operation corresponds to *implicit* intersection.

Definition 18 (Predicate environments). 1. A *predicate statement* is a construction of the form $\mathsf{e}:\mathrm{C} : \phi$, where e is an LJ expression, C is a class type and ϕ is a predicate. The expression e is called the *subject*, C is called the *type conclusion*, and ϕ is called the *predicate conclusion* of the statement.
2. A *predicate environment*, Π, is a set of predicate statements with term variables as subjects. As for type environment, we do not require the term variables to be distinct. Again, this is only a property of well formed environments, defined below.
3. As for type environments, we use the abbreviation $\Pi, x:\mathrm{C} : \phi$ to represent $\Pi \cup \{x:\mathrm{C} : \phi\}$. Similarly, we write Π, Π' to represent $\Pi \cup \Pi'$.
4. We write $x \in \Pi$ whenever there are C, ϕ such that $x:\mathrm{C} : \phi \in \Pi$.
5. We say the a predicate environment Π is an *object* predicate environment when the predicate conclusion of each statement Π is an object predicate that does not contain $\top$.
6. We extend the subpredicate relation to predicate environments as follows:

$$\Pi \trianglelefteq \Pi' \ \Leftrightarrow \ \forall\, x \in \Pi \ [\ x:\mathrm{C} : \phi \in \Pi \Rightarrow x:\mathrm{C} : \phi' \in \Pi' \ \&\ \phi \trianglelefteq \phi' \ ]$$

The idea behind the predicates is to make a statement on the execution of an expression: for example, if $\langle f : \phi \rangle$ is used to derive a predicate for the typed expression e:C, then the field f will be invoked when running e; to enforce this, we define:

Definition 19. For a type C, we define its *language*, $\mathcal{L}(\mathrm{C})$, as follows:

1. $\top \in \mathcal{L}(\mathrm{C})$ and $\langle \epsilon \rangle \in \mathcal{L}(\mathrm{C})$.
2. If $\psi \in \mathcal{L}(\mathrm{C})$, and there exists $m \in \mathcal{M}\ (\mathcal{EC}, \mathrm{C})$ such that $\mathcal{MT}(\mathcal{EC}, \mathrm{C}, m) = \overrightarrow{\mathrm{C}_n} \to \mathrm{D}$, $\phi_i \in \mathcal{L}(\mathrm{C}_i)$ for all $i \in \overline{n}$, and $\sigma \in \mathcal{L}(\mathrm{D})$, then $\langle m : \psi :: \overrightarrow{\phi_n} \to \sigma \rangle \in \mathcal{L}(\mathrm{C})$.
3. If $f \in \mathcal{F}\ (\mathcal{EC}, \mathrm{C})$, $\mathcal{FT}(\mathcal{EC}, \mathrm{C}, f) = \mathrm{D}$, and $\phi \in \mathcal{L}(\mathrm{D})$, then $\langle f:\phi \rangle \in \mathcal{L}(\mathrm{D})$.
4. If $\phi, \psi \in \mathcal{L}(\mathrm{C})$, then $\phi \sqcup \psi \in \mathcal{L}(\mathrm{C})$.

Definition 20 (Well-formed predicate environments). We say that a predicate environment Π is *well formed* with respect to some execution context $\mathcal{EC}$, when the statements in Π all have distinct variables as subjects, and the type conclusion of each statement is a valid class type with respect to $\mathcal{EC}$, and the predicate is an element of the language of that class type. This notion is formalised through the following judgements:

$$\frac{\vdash \mathcal{EC}}{\mathcal{EC} \vdash \emptyset} \qquad\qquad \frac{\mathcal{EC} \vdash \Pi \quad \vdash_{\mathcal{EC}} \mathrm{C}}{\mathcal{EC} \vdash \Pi, x:\mathrm{C} : \phi}\ (\phi \in \mathcal{L}(\mathrm{C}) \ \&\ \neg\exists\, \mathrm{D}, \phi'\ [\ x:\mathrm{D} : \phi' \in \Pi\ ])$$

$$[\text{P-NULL}]: \frac{\mathcal{EC} \vdash \Pi \quad \vdash_{\mathcal{EC}} \mathrm{C}}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} (\mathrm{C})\,\mathtt{null}{:}\mathrm{C} : \langle\epsilon\rangle} \qquad [\text{P-VAR}]: \frac{\mathcal{EC} \vdash \Pi}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} x{:}\mathrm{C} : \phi} (x{:}\mathrm{C} : \phi \in \Pi \;\&\; \phi \in \mathcal{L}(\mathrm{C}))$$

$$[\text{P-NEWO}]: \frac{\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}}){:}\mathrm{C}}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}}){:}\mathrm{C} : \langle\epsilon\rangle} \qquad [\text{P-FLD}]: \frac{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{D} : \langle f : \phi\rangle}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}.f{:}\mathrm{C} : \phi} (\mathcal{FT}(\mathcal{EC}, \mathrm{D}, f) = \mathrm{C})$$

$$[\text{P-ASS}_1]: \frac{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \langle f : \psi\rangle \quad \Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}'{:}\mathrm{D} : \phi}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}.f = \mathsf{e}'{:}\mathrm{C} : \langle f : \phi\rangle} (\mathcal{FT}(\mathcal{EC}, \mathrm{C}, f) = \mathrm{D})$$

$$[\text{P-ASS}_2]: \frac{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \langle l_i : \tau_i\ ^{i\in\underline{n}}\rangle \quad \widehat{\Pi} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}'{:}\mathrm{D}}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}.f = \mathsf{e}'{:}\mathrm{C} : \langle l_i : \tau_i\ ^{i\in\underline{n}}\rangle} (f \notin \vec{l}\ \&\ \mathcal{FT}(\mathcal{EC}, \mathrm{C}, f) = \mathrm{D})$$

$$[\text{P-INVK}]: \frac{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{D} : \langle m : \psi :: \vec{\phi} \to \sigma\rangle \quad \Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}_i{:}\mathrm{C}_i : \phi_i \;\; (\forall i \in \overline{n}) \quad \Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{D} : \psi}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}.m(\vec{\mathsf{e}}){:}\mathrm{C} : \sigma}$$
$$(\mathcal{MT}(\mathcal{EC}, \mathrm{D}, m) = \vec{\mathrm{C}} \to \mathrm{C})$$

$$[\text{P-NEWF}]: \frac{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}_j{:}\mathrm{C}_j : \phi \quad \widehat{\Pi} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathrm{C}_i{:} \quad (\forall i \in \overline{n}\,[\, i \neq j\,])}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}}){:}\mathrm{C} : \langle f_j : \phi\rangle}$$
$$(\mathcal{F}(\mathcal{EC}, \mathrm{C}) = \vec{f}\ \&\ j \in \overline{n}\ \&\ \forall\, i \in \overline{n}\,[\, \mathcal{FT}(\mathcal{EC}, \mathrm{C}, f_i) = \mathrm{C}_i\,])$$

$$[\text{P-NEWM}]: \frac{\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}}){:}\mathrm{C} \quad \Pi' \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}_0{:}\mathrm{D} : \sigma}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}}){:}\mathrm{C} : \langle m : \psi :: \overrightarrow{\phi_{n'}} \to \sigma\rangle}$$
$$\left(\begin{array}{l}\mathcal{MT}(\mathcal{EC}, \mathrm{C}, m) = \overrightarrow{\mathrm{C}_{n'}} \to \mathrm{D}\ \&\ \mathcal{M}b\,(\mathcal{EC}, \mathrm{C}, m) \\ \quad = (\overrightarrow{x_{n'}}, \mathsf{e}_0)\ \&\ \Pi' = \{\overrightarrow{x{:}\mathrm{C} : \phi_{n'}}, \mathtt{this}{:}\mathrm{C} : \psi\}\end{array}\right)$$

$$[\text{P-SUBT}]: \frac{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}' : \phi}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi} (\mathrm{C}' <:_{\mathcal{EC}} \mathrm{C}\ \&\ \phi \in \mathcal{L}(\mathrm{C})) \qquad [\text{P-TOP}]: \frac{\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \top}$$

$$[\text{P-JOIN}]: \frac{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi_i \quad (\forall i \in \overline{n})}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \sqcup\, \overrightarrow{\phi_n}} \qquad [\text{P-SEQ}]: \frac{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \psi}{\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi} (\psi \trianglelefteq \phi)$$

Fig. 2. Predicate Assignment Rules

Notice that, by simple induction on the derivation of $\mathcal{EC} \vdash \Pi$, it is easy to see that $\mathcal{EC} \vdash \Pi'$ for any $\Pi' \subseteq \Pi$.

Definition 21 (Environment conversion). The notation $\widehat{\Pi}$ denotes the type environment obtained by discarding the predicate conclusions from the statements in Π:

$$\widehat{\Pi} \triangleq \{\, x{:}\mathrm{C} \mid x{:}\mathrm{C} : \phi \in \Pi \,\}$$

Definition 22 (Predicate assignment). Predicate assignment is a ternary relation between execution contexts, predicate environments and predicate statements, written as $\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi$. The relation $\vdash^{\mathrm{P}}$ is defined using the natural deduction system of Figure 2.

We can see the predicate system as a Hoare-style system of pre and post conditions. For example, for rule (P-INVK), the rule expresses that if the arguments $\overrightarrow{\mathsf{e}_i}$ for the method call satisfy, respectively, $\overrightarrow{\phi_i}$, and the object satisfies ψ, that then the method that is going to be called will satisfy σ, giving an annotation like:

:: pre: each e_i satisfies ϕ_i & e satisfies ψ
e.$m(\vec{\mathsf{e}_i})$
:: post: σ

The results of this paper provide the semantic underpinning for such a system.

4.1 Properties of the Type Systems

We now present some properties of the type systems defined above. We begin by showing that the set of expressions typeable by the class type system is exactly the same as the set of expressions typeable by the predicate system; because of space restrictions, proofs are omitted.

Theorem 23. $\exists\, \phi\ [\ \Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi\ ] \Leftrightarrow \widehat{\Pi} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}$

The following results are a crucial part of the proof for subject expansion. The first states that a label that occurs in a predicate is visible in the type, and the second that predicate assignment is closed for subtyping, as long as the predicate used is in the language of the supertype.

Theorem 24. *1. If* $\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi$, *then* $\phi \in \mathcal{L}(\mathrm{C})$.
2. If $\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi$, $\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}'$, *and* $\phi \in \mathcal{L}(\mathrm{C}')$, *then* $\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}' : \phi$.

The main result of this paper is now formulated be the next two theorems: first, both types and predicates are preserved by reduction:

Theorem 25. *1.* $\vdash^{\mathrm{T}}_{\mathcal{EC}} \diamond$ & $\Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}$ & $\mathsf{e} \to_{\mathcal{EC}} \mathsf{e}' \Rightarrow \Gamma \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}'{:}\mathrm{C}$
2. $\vdash^{\mathrm{T}}_{\mathcal{EC}} \diamond$ & $\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi$ & $\mathsf{e} \to_{\mathcal{EC}} \mathsf{e}' \Rightarrow \Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}'{:}\mathrm{C} : \phi$

Predicates are preserved by expansion:

Theorem 26. $\vdash^{\mathrm{T}}_{\mathcal{EC}} \diamond$ & $\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}'{:}\mathrm{C} : \phi$ & $\mathsf{e} \to_{\mathcal{EC}} \mathsf{e}'$ & $\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} \Rightarrow \Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi$

4.2 Characterisation of Expressions

As with the intersection type system for the λ-calculus, the subject expansion result for the predicate system allows us to characterise the behaviour of LJ expressions by the predicates that we can assign to them.

We first see that (using an object predicate environment) any LJ expression that terminates in an object can be assigned a non-trivial predicate that *does not* include $\top$.

Theorem 27. *If there exists* Π, C *such that* $\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C}$ *and* $\mathsf{e} \to^* \mathsf{o}$, *then there exists* ϕ *such that* $\phi \neq \top$ *and* $\Pi \vdash^{\mathrm{P}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi$.

We can illustrate this property by the following program:

$$\begin{array}{l}\mathcal{EC} = \texttt{class C extends Object}\\ \qquad \{\ \texttt{C m}(\,)\ \{\ \texttt{this.m}(\,)\ \}\ \ \}\\ \texttt{e}\ \ = (\texttt{new C}\ (\,)).\texttt{m}(\,)\end{array}$$

The expression e runs to itself, thus $\texttt{e} \rightarrow_{\mathcal{EC}} \texttt{e} \rightarrow_{\mathcal{EC}} \ldots$ ad infinitum. Notice that $\emptyset \vdash^{\mathrm{T}}_{\mathcal{EC}} \texttt{e:C}$, and so by rule P-TOP we have $\emptyset \vdash^{\mathrm{P}}_{\mathcal{EC}} \texttt{e:C} : \top$. However, since e only reduces to itself, a consequence of the approximation result, $\top$ is the *only* predicate that we can assign to e.

Now, consider the following LJ program:

$$\begin{array}{l}\texttt{class C extends Object}\\ \ \{\ \texttt{C f}\\ \ \ \texttt{C m}(\,)\ \{\ \texttt{this.f} = \texttt{this.m}(\,)\ \}\\ \ \}\\ \texttt{e} = (\texttt{new C}\ ((\texttt{C})\,\texttt{null})).\texttt{m}(\,)\end{array}$$

which results in this sequence of reductions:

$$\begin{array}{lll}
 & \texttt{e} & \equiv\ (\texttt{new C}\ ((\texttt{C})\,\texttt{null})).\texttt{m}(\,) \\
\rightarrow_{\mathcal{EC}} & & (\texttt{new C}\ ((\texttt{C})\,\texttt{null})).\texttt{f} = (\texttt{new C}\ ((\texttt{C})\,\texttt{null})).\texttt{m}(\,) \\
\rightarrow_{\mathcal{EC}} & \texttt{e}_1 & \equiv\ \texttt{new C}\ ((\texttt{new C}\ ((\texttt{C})\,\texttt{null})).\texttt{m}(\,)) \\
\rightarrow_{\mathcal{EC}} & & \texttt{new C}\ ((\texttt{new C}\ ((\texttt{C})\,\texttt{null})).\texttt{f} = (\texttt{new C}\ ((\texttt{C})\,\texttt{null})).\texttt{m}(\,)) \\
\rightarrow_{\mathcal{EC}} & \texttt{e}_2 & \equiv\ \texttt{new C}\ (\texttt{new C}\ ((\texttt{new C}\ ((\texttt{C})\,\texttt{null})).\texttt{m}(\,))) \\
\rightarrow^{*}_{\mathcal{EC}} & \texttt{e}_3 & \equiv\ \texttt{new C}\ (\texttt{new C}\ (\texttt{new C}\ ((\texttt{new C}\ ((\texttt{C})\,\texttt{null})).\texttt{m}(\,)))) \qquad \textit{etc}
\end{array}$$

Thus, this expression constructs an ever increasingly nested object. Observe that $\emptyset \vdash^{\mathrm{T}}_{\mathcal{EC}} \texttt{e:C}$ and so by rule P-TOP it follows that $\emptyset \vdash^{\mathrm{P}}_{\mathcal{EC}} \texttt{e:C} : \top$. Given this, we can assign the following predicates to the sequence of expressions $\vec{\texttt{e}}$:

$$\begin{array}{l}
\emptyset \vdash^{\mathrm{P}}_{\mathcal{EC}} \texttt{e}_1\texttt{:C} : \langle \texttt{f}{:}\top \rangle \\
\emptyset \vdash^{\mathrm{P}}_{\mathcal{EC}} \texttt{e}_2\texttt{:C} : \langle \texttt{f}{:}\langle \texttt{f}{:}\top \rangle \rangle \\
\emptyset \vdash^{\mathrm{P}}_{\mathcal{EC}} \texttt{e}_3\texttt{:C} : \langle \texttt{f}{:}\langle \texttt{f}{:}\langle \texttt{f}{:}\top \rangle \rangle \rangle \quad \textit{etc}
\end{array}$$

Then, by subject expansion, we can assign all of these predicates to the expression e itself. Thus, we can assign a non-trivial predicate to e, but it must contain $\top$, suggesting that there is non-termination somewhere.

4.3 Expressiveness

A final point that we can make concerns the expressiveness of the predicate system over the type system. A result of the type system (and similarly of the type systems of FJ, MJ and Java itself) is that if an expression is typeable, then executing the expression will not result in any illegal field accesses or method invocations. In other words, whenever a field is accessed, or a method invoked, such a field or method will always exist in the receiving object. One thing that *may* happen, however, is a null reference exception. This occurs when a field is accessed or a method invoked on a null object. The type

system does not distinguish between the types of null objects and the types of non-null objects; thus it cannot determine when such a mismatch will occur. The predicate system, on the other hand, *does* make such a distinction: the only non-trivial predicate that null objects may be assigned is the empty predicate $\langle\epsilon\rangle$. As such, a field access or a method invocation on a null object cannot be assigned any predicate other than $\top$, since the premise for such a predicate assignment is that the receiver have an appropriate non-empty object predicate. Thus, again by subject expansion, it follows that the execution of any expression which can be assigned a non-trivial predicate will not result in a null reference exception.

5 Approximants for Lightweight Java

In the context of formal calculi, termination - all computations finish - is an much studied problem, and often type theory makes this result achievable. For example, for the λ-calculus [12], it is well-known that there exists non-terminating terms, but that (using simple type assignment) all typeable term terminate. In the context of intersection types, on which our predicates are based, this property holds, but with modifications; in fact, using intersection types, it is possible to show:

- If $\Gamma \vdash M : \sigma$ and $M =_\beta N$, then $\Gamma \vdash N : \sigma$.
- $\Gamma \vdash M : \sigma$ and $\sigma \neq \top$, if and only if M has a head-normal form.
- $\Gamma \vdash M : \sigma$ and $\top$ does not occur in Γ and σ, if and only if M has a normal form.
- $\Gamma \vdash M : \sigma$ and $\top$ is not used at all, if and only if M is strongly normalisable.

Another, perhaps less known property that holds for the intersection type assignment system for the λ-calculus is the approximation theorem:

$$\Gamma \vdash M : \sigma \Leftrightarrow \exists\, A \in \mathcal{A}(M)\, [\, \Gamma \vdash A : \sigma\,]$$

This result is based on the notion of approximant for λ-terms, as first presented by C. Wadsworth in [61], which is defined as follows.

Definition 28. 1. The set of $\lambda\Omega$-*terms*[2] is defined as the set of λ-terms by adding the constant Ω to the syntax.
2. The notion of reduction $\rightarrow_{\beta\Omega}$ $\lambda\Omega$-*terms* is defined as $\rightarrow_\beta$, extended by:

$$\lambda x.\Omega \;\rightarrow_{\beta\Omega}\; \Omega \qquad \Omega M \;\rightarrow_{\beta\Omega}\; \Omega.$$

3. The set of *normal forms* with respect to $\rightarrow_{\beta\Omega}$ is the set $\mathcal{A}$ of $\lambda\Omega$-*normal forms* or *approximate normal forms*, ranged over by A, defined by:

$$A \;::=\; \Omega \mid \lambda x.A\ (A \neq \Omega) \mid x A_1 \ldots A_n\ (n \geq 0)$$

We can view Ω as hiding a place where (infinite) reduction might take place.

[2] Ω is the symbol originally used in [61]; more common now is to, as [12], use the symbol $\bot$; since this could be confused to be a predicate, we have opted for the old notation.

Definition 29 (Approximants). 1. The partial order $\sqsubseteq$ is defined as the smallest pre-order such that:

$$\begin{array}{rcl} \Omega & \sqsubseteq & M \\ M \sqsubseteq M' & \Rightarrow & \lambda x.M \sqsubseteq \lambda x.M' \\ M_1 \sqsubseteq M_1' \;\&\; M_2 \sqsubseteq M_2' & \Rightarrow & M_1 M_2 \sqsubseteq M_1' M_2' \end{array}$$

2. If $A \sqsubseteq M$, then A is called a *direct approximant* of M.
3. The relation $\underset{\sim}{\sqsubseteq}$ is defined by: $A \underset{\sim}{\sqsubseteq} M \Leftrightarrow \exists N\,[\, M \twoheadrightarrow_\beta N \;\&\; A \sqsubseteq N\,]$.
4. If $A \underset{\sim}{\sqsubseteq} M$, then A is called an *approximant* of M.
5. $\mathcal{A}(M) = \{\, A \in \mathcal{A} \mid A \underset{\sim}{\sqsubseteq} M \,\}$.

Since it is easy to show that, if $M \twoheadrightarrow_\beta N$, then, for all A, if $A \sqsubseteq M$, then $A \sqsubseteq N$, but not vice versa, we can see the approximants of a term M as representing the dynamics of running M: if $A \sqsubseteq M$ is not Ω, then A exposes some structure (either an abstraction or a head-variable) and the above property then states that this structure will not disappear while continuing the execution; it is the output of (running) M.

Definition 30. 1. The partial mapping $\sqcup$ (*join*) is defined by:

$$\begin{array}{rcl} \Omega \sqcup M = M \sqcup \Omega & = & M \\ x \sqcup x & = & x \\ (\lambda x.M) \sqcup (\lambda x.N) & = & \lambda x.(M \sqcup N) \\ (M_1 M_2) \sqcup (N_1 N_2) & = & (M_1 \sqcup N_1)\,(M_2 \sqcup N_2) \end{array}$$

2. If $M \sqcup N$ is defined, then M and N are called *compatible*.

Now $\sqcup$ acts as least upper bound of compatible terms. The set of approximants of M corresponds to the finite, rooted segments of $BT(M)$, the Böhm tree of M, a tree that represents the (possible infinite) normal form of M (see [12]).

$$\bigsqcup\{\, A \mid A \in \mathcal{A}(M) \,\} \;\sim\; BT(M)$$

where (possibly infinite) subtrees are replaced by Ω; in fact, we can show that $M =_\beta N$ implies $\mathcal{A}(M) = \mathcal{A}(N)$. This observation immediately gives that we can define a model for the λ-calculus by interpreting terms by their approximants.

The approximation result stated above now directly links types with semantics, and generalises the standard subject reduction result, which states that types are preserved by reduction. To put the approximation result into words, it states that, for every typeable term M, during the execution of M some intermediate result will be reached that we can type in exactly the same way, perhaps masking some parts with $\top$: so intersection types allow a look-ahead over execution.

We will now define a notion of approximants for LJ, and link the predicates we assign an expressions e to its approximants in the same way in the next section. First we define approximants.

Definition 31 (APPROXIMATE EXPRESSIONS). 1. We extend the syntax of LJ expressions with an element Ω, and define *approximate* expressions using the following grammar:

$$\mathsf{a} \;::=\; x \mid \Omega \mid (\mathrm{C})\,\mathtt{null} \mid \mathsf{a}.f \mid \mathsf{a}.f = \mathsf{a}' \mid \mathsf{a}.m(\vec{\mathsf{a}}) \mid \mathsf{new}\,\mathrm{C}(\vec{\mathsf{a}})$$

2. We extend the reduction relation to approximate expressions with the following rules:

$$\Omega.f \to_{\mathscr{EC}} \Omega \quad \Omega.f = \mathsf{a} \to_{\mathscr{EC}} \Omega \quad \Omega.m(\vec{\mathsf{a}}) \to_{\mathscr{EC}} \Omega$$

The normal forms of approximate expressions (or *approximate normal forms*) with respect to this extended reduction relation are defined by the following grammar:

$$\begin{array}{rl} \mathsf{A} ::= & x \mid \Omega \mid (\mathrm{C})\,\mathtt{null} \mid \mathsf{new}\,\mathrm{C}(\vec{\mathsf{A}}) \mid \\ & \mathsf{A}.f \mid \mathsf{A}.f = \mathsf{A}' \mid \mathsf{A}.m(\vec{\mathsf{A}}) \qquad (\mathsf{A},\mathsf{A}' \neq \Omega \;\&\; \mathsf{A},\mathsf{A}' \neq \mathsf{new}\,\mathrm{C}(\vec{\mathsf{A}})) \end{array}$$

3. The rules for type assignment are extended to type approximate expressions by allowing expressions to contain Ω; this implies that, if a contains Ω, then $\top$ is used to cover a subterm of a with $\top$. Moreover, the only predicate assignable to Ω (and indeed to field and method invocations on Ω) is $\top$.

Definition 32 (APPROXIMANTS). 1. The relation $\sqsubseteq$ over approximate expressions is defined as the smallest pre-order satisfying:

$$\begin{array}{rcl} \Omega & \sqsubseteq & \mathsf{e} \\ \mathsf{e} \sqsubseteq \mathsf{e}' & \Rightarrow & \mathsf{e}.f \sqsubseteq \mathsf{e}'.f \\ \mathsf{e}_1 \sqsubseteq \mathsf{e}'_1 \;\&\; \mathsf{e}_2 \sqsubseteq \mathsf{e}'_2 & \Rightarrow & \mathsf{e}_1.f = \mathsf{e}_2 \sqsubseteq \mathsf{e}'_1.f = \mathsf{e}'_2 \\ \mathsf{e} \sqsubseteq \mathsf{e}' \;\&\; \forall\, i \in \overline{n}\ [\ \mathsf{e}_i \sqsubseteq \mathsf{e}'_i\ ] & \Rightarrow & \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}) \sqsubseteq \mathsf{e}'.m(\overrightarrow{\mathsf{e}'_n}) \\ \forall\, i \in \overline{n}\ [\ \mathsf{e}_i \sqsubseteq \mathsf{e}'_i\ ] & \Rightarrow & \mathsf{new}\,\mathrm{C}(\overrightarrow{\mathsf{e}_n}) \sqsubseteq \mathsf{new}\,\mathrm{C}(\overrightarrow{\mathsf{e}'_n}) \end{array}$$

If $\mathsf{A} \sqsubseteq \mathsf{e}$ then we say that A is a *direct approximant* of e (notice that this notion is independent of the evaluation context).

2. We define the binary relation $\underset{\sim}{\sqsubseteq}_{\mathscr{EC}}$ on approximate normal forms and expressions as follows: $\mathsf{A} \underset{\sim}{\sqsubseteq}_{\mathscr{EC}} \mathsf{e} \Leftrightarrow \exists\, \mathsf{e}'\ [\ \mathsf{e} \to^*_{\mathscr{EC}} \mathsf{e}' \;\&\; \mathsf{A} \sqsubseteq \mathsf{e}'\ ]$. If $\mathsf{A} \underset{\sim}{\sqsubseteq}_{\mathscr{EC}} \mathsf{e}$ then we say that A is an *approximant* of e.
3. We define: $\mathcal{A}_{\mathscr{EC}}(\mathsf{e}) = \{\, \mathsf{A} \mid \mathsf{A} \underset{\sim}{\sqsubseteq}_{\mathscr{EC}} \mathsf{e} \,\}$.

We can now show the following properties:

Lemma 33. *1. If* $\mathsf{e} \to^*_{\mathscr{EC}} \mathsf{e}'$ *and* $\mathsf{A} \sqsubseteq \mathsf{e}$, *then* $\mathsf{A} \sqsubseteq \mathsf{e}'$.
2. If $\mathsf{e} \to^*_{\mathscr{EC}} \mathsf{e}'$ *then* $\mathcal{A}_{\mathscr{EC}}(\mathsf{e}) = \mathcal{A}_{\mathscr{EC}}(\mathsf{e}')$.

With the second of these results, we have an approximation semantics for LJ: $[\![\mathsf{e}]\!]^{\mathcal{A}}_{\mathscr{EC}} = \mathcal{A}_{\mathscr{EC}}(\mathsf{e})$.

6 Approximation Result

We define a family of predicates that assert approximation of an expression with a given predicate type:

Definition 34. $Appr_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \phi) \;\Leftrightarrow\; \widehat{\Pi} \vdash^{\mathrm{T}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C} \;\&\; \exists\, \mathsf{A} \in \mathcal{A}_{\mathscr{EC}}(\mathsf{e})\ [\ \Pi \vdash^{\mathrm{P}}_{\mathscr{EC}} \mathsf{A}{:}\mathrm{C} : \phi\ ]$

In this section, we will show the approximation result for our notion of predicate assignment for LJ, that states: $\Pi \vdash^{\mathrm{p}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C} : \phi \Leftrightarrow \mathit{Appr}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \phi)$. This theorem states that for every expression e to which we can assign the predicate ϕ, there exists an approximant of e to which the same predicate can be assigned. This theorem will allow us to characterise expressions which have a head normal form by their assignable predicates.

First, we show that predicate assignment is upward closed for $\sqsubseteq$:

Theorem 35. $\Pi \vdash^{\mathrm{p}}_{\mathscr{EC}} \mathsf{a}{:}\mathrm{C} : \phi \;\&\; \widehat{\Pi} \vdash^{\mathrm{T}}_{\mathscr{EC}} \mathsf{a}'{:}\mathrm{C}' \;\&\; \mathsf{a} \sqsubseteq \mathsf{a}' \Rightarrow \Pi \vdash^{\mathrm{p}}_{\mathscr{EC}} \mathsf{a}'{:}\mathrm{C}' : \phi$.

Notice, in particular, this result holds when $\mathrm{C} = \mathrm{C}'$.

We define a subset of expressions that are those that start with a variable; we need this notion because we want to show that all variables are computable of any type and predicate. In order to show that, using of course the computability predicate, being defined by induction on the structure of predicates, we have to consider arbitrary sequences of field of method invocations, or field overrides; the set of neutral terms is, therefore, the set of variables, 'closed' for those calls.

Definition 36 (NEUTRAL EXPRESSIONS). *Neutral expressions* are defined by the following grammar:

$$\mathsf{n} ::= x \mid \mathsf{n}.f \mid \mathsf{n}.f = \mathsf{e} \mid \mathsf{n}.m(\vec{\mathsf{e}})$$

Notice that neutral expressions are not, in general, in normal form, but any reductions that can take place will be performed on the arguments to method invocations or field assignments. Thus, the 'externally visible' structure of the expression remains constant, as stated by the following lemma:

Lemma 37. *1.* $\mathsf{A} \in \mathcal{A}_{\mathscr{EC}}(\mathsf{n}) \Rightarrow \mathsf{A}.f \in \mathcal{A}_{\mathscr{EC}}(\mathsf{n}.f)$.
2. $\mathsf{A} \in \mathcal{A}_{\mathscr{EC}}(\mathsf{n}) \;\&\; \mathsf{A}' \in \mathcal{A}_{\mathscr{EC}}(\mathsf{e}) \Rightarrow \mathsf{A}.f = \mathsf{A}' \in \mathcal{A}_{\mathscr{EC}}(\mathsf{n}.f = \mathsf{e})$.
3. $\mathsf{A} \in \mathcal{A}_{\mathscr{EC}}(\mathsf{n}) \;\&\; \forall\, i \in \overline{m}\, [\, \mathsf{A}_i \in \mathcal{A}_{\mathscr{EC}}(\mathsf{e}_i)\,] \Rightarrow \mathsf{A}.m(\overrightarrow{\mathsf{A}_m}) \in \mathcal{A}_{\mathscr{EC}}(\mathsf{n}.m(\overrightarrow{\mathsf{e}_m}))$.

We will use Taits' proof method [58] involving a *computability predicate*, which we now define.

Definition 38 (COMPUTABILITY PREDICATE). We define a family of computability predicates, over execution contexts $\mathscr{EC}$, inductively as follows:

$$\begin{aligned}
\mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \top) &\Leftrightarrow \mathit{Appr}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \top) \\
\mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \langle \epsilon \rangle) &\Leftrightarrow \mathit{Appr}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \langle \epsilon \rangle) \\
\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C} \;\&\; \mathcal{FT}(\mathscr{EC}, \mathrm{C}, f) = \mathrm{D} \Rightarrow & \\
(\mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \langle f : \top \rangle) &\Rightarrow \mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}.f{:}\mathrm{D}, \top)) \\
\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C} \;\&\; \mathcal{FT}(\mathscr{EC}, \mathrm{C}, f) = \mathrm{D} \Rightarrow & \\
(\mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \langle f : \sigma \rangle) &\Leftrightarrow \mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}.f{:}\mathrm{D}, \sigma)) \\
\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C} \;\&\; \mathcal{MT}(\mathscr{EC}, \mathrm{C}, m) = \overrightarrow{\mathrm{C}_n} \to \mathrm{D} \Rightarrow & \\
(\mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \langle m : \psi :: \overrightarrow{\phi_n} \to \sigma \rangle) &\Leftrightarrow \\
(\mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \psi) \;\&\; \forall\, i \in \overline{n}\, [\, \mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}_i{:}\mathrm{C}_i, \phi_i)\,] & \\
\Rightarrow \mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}){:}\mathrm{D}, \sigma))) & \\
\mathrm{D} <:_{\mathscr{EC}} \mathrm{C} \Rightarrow \quad (\mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \phi) &\Leftrightarrow \mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{D}, \phi)) \\
\mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \sqcup \overrightarrow{\phi_{i\overline{n}}}) &\Leftrightarrow \forall\, i \in \overline{n}\, [\, \mathit{Comp}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \phi_i)\,]
\end{aligned}$$

We now show that computability applies approximation, and that approximable *neutral* terms are computable:

Theorem 39. *1.* $Appr_{\mathscr{EC}}(\Pi, \mathsf{n}{:}\mathrm{C}, \phi) \Rightarrow Comp_{\mathscr{EC}}(\Pi, \mathsf{n}{:}\mathrm{C}, \phi)$.
2. $Comp_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \phi) \Rightarrow Appr_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \phi)$.

A corollary of this theorem is that variables are computable of any predicate which is assignable to them.

Corollary 40. $\{\, x{:}\mathrm{C}{:}\phi \,\} \vdash^{\mathrm{P}}_{\mathscr{EC}} x{:}\mathrm{C} : \phi \Rightarrow Comp_{\mathscr{EC}}(\{\, x{:}\mathrm{C}{:}\phi \,\}, x{:}\mathrm{C}, \phi)$

The next step is to formulate a *replacement* lemma, which states that if we replace all the variables in a predicable expression with expressions computable of appropriate predicates, then we obtain a computable expression.

Lemma 41 (Replacement Lemma). If $\Pi \vdash^{\mathrm{P}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C} : \phi$, and there exists $\Pi', \vec{\mathsf{e}_i}$ such that, for all $x_i{:}\mathrm{C}_i{:}\phi_i \in \Pi$ we have $Comp_{\mathscr{EC}}(\Pi', \mathsf{e}_i{:}\mathrm{C}_i, \phi_i)$, then $Comp_{\mathscr{EC}}(\Pi', \mathsf{e}[\overrightarrow{\mathsf{e}_i / x_i}]{:}\mathrm{C}, \phi)$.

It is worthwhile to note that this technique, in the context of the λ-calculus, requires the proof for the statement $Comp(\Gamma, M[N/x], \sigma) \Leftrightarrow Comp(\Gamma, (\lambda x.M)N, \sigma)$; this is needed mainly because the λ-calculus has *abstraction*, accompanied by the type assignment rule

$$\frac{\Gamma, x{:}A \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}$$

where the subderivation has a larger context. Notice that none of the rules for LJ have this particular feature, so we do not have the added complexity of having to reason over the replacements used.

A corollary of the replacement lemma will be that if an expression can be assigned a predicate ϕ, then it is computable of that predicate.

Corollary 42 (Typeability implies Computability). $\Pi \vdash^{\mathrm{P}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C} : \phi \Rightarrow Comp_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, \phi)$.

Combining this with Theorem 40 gives that if an expression can be assigned a predicate ϕ then it has an approximant which can also be assigned ϕ.

Theorem 43 (Typeability implies Approximability). If $\Pi \vdash^{\mathrm{P}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C} : \phi$ then $\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C}$ and there exists $\mathsf{A} \in \mathcal{A}_{\mathscr{EC}}(\mathsf{e})$ such that $\Pi \vdash^{\mathrm{P}}_{\mathscr{EC}} \mathsf{A}{:}\mathrm{C} : \phi$.

7 Characterisation of head-normalisation

In this section, we will show the main result of this paper:

If $\Pi \vdash^{\mathrm{P}}_{\mathscr{EC}} \mathsf{e}{:}\mathrm{C} : \phi$, and $\phi \neq \top$, then e has a head-normal form.

and show that this follows from the approximation result.

Definition 44 (HEAD NORMAL FORMS). The set $\mathcal{H}$ of expressions in *head-normal form* is defined by:

$$\begin{array}{rl} \mathsf{H} ::= & x \mid (\mathrm{C})\,\mathtt{null} \mid \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}}) \mid \\ & \mathsf{H}.f \mid \mathsf{H}.f = \mathsf{e} \mid \mathsf{H}.m(\vec{\mathsf{e}}) \quad (\mathsf{H} \neq \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}})) \end{array}$$

Lemma 45. *1. If* $\mathsf{A} \sqsubseteq \mathsf{e}$, *then* $\mathsf{e} \in \mathcal{H}$.
2. If $\mathsf{H} \in \mathcal{H}$, *then there exist* Π, C, *and* ϕ *such that* $\Pi \vdash^{\mathrm{P}}_{\mathscr{L}} \mathsf{H}{:}\mathrm{C} : \phi$.

We can now show the head-normalisation result:

Theorem 46. *Assume there exists* Π, *and* C *such that* $\widehat{\Pi} \vdash^{\mathrm{T}}_{\mathscr{L}} \mathsf{e}{:}\mathrm{C}$. *Then: there exist* ϕ *such that* $\Pi \vdash^{\mathrm{P}}_{\mathscr{L}} \mathsf{e}{:}\mathrm{C} : \phi$, *and* $\phi \neq \top$ *iff* e *has a head-normal form.*

8 About normalisation

Another well-known result for intersection types in the context of the λ-calculus is the characterisation of *normalisation* via: $\Gamma \vdash M : \sigma$ and $\top$ not in Γ and σ, if and only if M has a normal form.

First, we define the notion of normal form with respect to $\rightarrow_{\mathscr{L}}$:

Definition 47 (NORMAL FORMS). The set $\mathcal{N}$ of expressions in *normal form* is defined by:

$$\begin{array}{rl} \mathsf{N} ::= & x \mid (\mathrm{C})\,\mathtt{null} \mid \mathsf{new}\,\mathrm{C}(\vec{\mathsf{N}}) \mid \\ & \mathsf{N}.f \mid \mathsf{N}.f = \mathsf{N}' \mid \mathsf{N}.m(\vec{\mathsf{N}}) \quad (\mathsf{N} \neq \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}})) \end{array}$$

As argued in [11], the characterisation of normalisation only holds in the context of *strong reduction*, and we cannot show it for the system we are considering here. This is not only because we can find normal forms that are only typeable with $\top$ (like $\mathtt{null}.f$), but also that a predicate need not show *all* fields that are visible in a type

Example 48. Take

```
class C extends Object
  { C f
    C g
    C m( ) { this . m( ) }
  }
e = new C ( (C) null, new C ( (C) null, (C) null) . m( ) )
```

Now we can derive $\emptyset \vdash^{\mathrm{P}}_{\mathscr{L}} \mathsf{e}{:}\mathrm{C} : \langle f{:}\langle\epsilon\rangle\rangle$, and $\emptyset \vdash^{\mathrm{P}}_{\mathscr{L}} \mathsf{e}{:}\mathrm{C} : \langle g{:}\top\rangle$. Notice that we have an infinite reduction from e:

$$\begin{array}{ll} \mathsf{new}\,\mathrm{C}\,((\mathrm{C})\,\mathtt{null}, \mathsf{new}\,\mathrm{C}\,((\mathrm{C})\,\mathtt{null}, (\mathrm{C})\,\mathtt{null})\,.\,\mathtt{m}(\,)\,) & \rightarrow_{\mathscr{L}} \\ \mathsf{new}\,\mathrm{C}\,((\mathrm{C})\,\mathtt{null}, \mathtt{this}\,.\,\mathtt{m}(\,)[\mathsf{new}\,\mathrm{C}\,((\mathrm{C})\,\mathtt{null}, (\mathrm{C})\,\mathtt{null})/\mathtt{this}]\,) & = \\ \mathsf{new}\,\mathrm{C}\,((\mathrm{C})\,\mathtt{null}, \mathsf{new}\,\mathrm{C}\,((\mathrm{C})\,\mathtt{null}, (\mathrm{C})\,\mathtt{null})\,.\,\mathtt{m}(\,)\,) & \rightarrow_{\mathscr{L}} \ldots \end{array}$$

although we can assign a predicate not containing $\top$.

So a characterisation of normalisation for the free reduction of this paper is not achievable. However, if we switch to lazy reduction, then the head-normal forms are exactly the normal forms, and the characterisation of head-normalisation and normalisation collapse onto one result, already show above.

Definition 49 (LAZY NORMAL FORMS). The set $\mathcal{N}_\ell$ of expressions in *lazy normal form* is defined by:

$$\begin{array}{rcl} \mathsf{N}_\ell & ::= & x \mid (\mathrm{C})\,\mathtt{null} \mid \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}}) \mid \\ & & \mathsf{N}_\ell.f \mid \mathsf{N}_\ell.f = \mathsf{e}' \mid \mathsf{N}_\ell.m(\vec{\mathsf{e}}) \;\; (\mathsf{N} \neq \mathsf{new}\,\mathrm{C}(\vec{\mathsf{e}})) \end{array}$$

Theorem 50. *When restricting to* lazy *evaluation, there exist* Π, C, *and* ϕ *such that* $\Pi \vdash^{\mathrm{p}}_{\mathcal{EC}} \mathsf{e}{:}\mathrm{C} : \phi$*, and* $\phi \neq \top$ *iff* e *has a lazy normal form.*

9 Conclusions and Future Work

There are many directions that future research in this area could take. One such avenue of investigation that would further cement the theoretical foundations of the class-based object oriented paradigm is to define an encoding of the Lambda Calculus in LJ. This has been done for the ς-calculus in [1]. This would demonstrate the expressive power and equivalence of LJ with the λ-calculus. On a broader note, semantic models for LJ, as well as other class-based calculi, could be developed. [10] addresses this issue for the ς-calculus, so it seems likely that a similar approach could be taken for LJ.

We have mentioned how LJ is a functional calculus and, as such, lacks imperative features like the ones included in MJ. A further step might be to add these features to LJ, and also extend the predicate system to handle them. It would be interesting to see if they can be subsumed easily into the predicate system, or whether the presence of side-effects will necessitate more drastic changes. Taking another lead from [15], we could also incorporate an effects system into our calculus. Again, one would hope that this extension would dovetail easily with the predicate system.

As was the case with intersection types for the λ-calculus, the predicates we allow for the construction of a filter model by interpreting expressions by their assignable predicates. We have also briefly touched on characterisation properties in Section 4.2. One natural progression of our work will be to fully investigate this characterisation of expressions, and show that all programs typeable in a system that excludes $\top$ will terminate.

Finally, the *predicate* approach of this paper opens the way for functional-style type-based abstract interpretation, like side-effect analysis via type-and-effect systems, and systems that analyse aliasing. We aim to extend our approach into that direction.

References

1. M. Abadi and L. Cardelli. *A Theory Of Objects.* Springer-Verlag, 1996.

2. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of java bytecode. In *Proceedings of* Formal Methods for Open Object-Based Distributed Systems (FMOODS'08)*, 2008*, *LNCS* 5051, pages 2–18, 2008.
3. S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, University of Nijmegen, 1993.
4. S. van Bakel. Intersection Type Assignment Systems. *Theoretical Computer Science*, 151(2):385–435, 1995.
5. S. van Bakel. Rank 2 Intersection Type Assignment in Term Rewriting Systems. *Fundamenta Informaticae*, 2(26):141–166, 1996.
6. S. van Bakel. Rank 2 Types for Term Graph Rewriting (Extended Abstract). In *Electronic Proceedings of International Workshop* Types in Programming *(TIP'02), Dagstuhl, Germany*, *ENTCS* 75, 2002.
7. S. van Bakel. The Heart of Intersection Type Assignment; Normalisation proofs revisited. *Theoretical Computer Science*, 398:82–94, 2008.
8. S. van Bakel and U. de'Liguoro. Logical semantics for the first order sigma calculus. In *Proceedings of Eighth Italian Conference on Theoretical Computer Science (ICTCS'03), Bertinoro, Italy*, *LNCS* 2841, pages 202–215, 2003.
9. S. van Bakel and U. de'Liguoro. Logical semantics for $\mathsf{fob}_{1<:\mu}$. In *Proceedings of Ninth Italian Conference on Theoretical Computer Science (ICTCS'05), Siena, Italy*, *LNCS* 3701, pages 66–80, 2005.
10. S. van Bakel and U. de'Liguoro. Logical Equivalence for Subtyping Object and Recursive Types. *Theory of Computing Systems*, 42(3):306–348, 2008.
11. S. van Bakel and M. Fernández. Normalisation, Approximation, and Semantics for Combinator Systems. *Theoretical Computer Science*, 290:975–1019, 2003.
12. H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, 1981.
13. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
14. P. N. Benton. *Strictness Analysis of Lazy Functional Programs*. PhD thesis, Computing Lab, University of Cambridge, 1993. Technical Report 309.
15. G. Bierman, M.J. Parkinson, and A. Pitts. MJ: An Imperative Core Calculus for Java and Java with Effects. Technical Report 563, University of Cambridge Computer Lab, 2003.
16. L. Cardelli. A Semantics of Multiple Inheritance. In *Proc. of the international symposium on Semantics of data types*, 51–67, 1984.
17. L. Cardelli and J.C. Mitchell. Operations on Records. In *Proceedings of the fifth international conference on Mathematical foundations of programming semantics*, 22–52, 1990.
18. G. Castagna and B.C. Pierce. Decidable bounded quantification. In *Proceedings of Symposium on Principles of Programming Languages (POPL'04)*, 151–162, 1994.
19. M. Coppo and M. Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-Calculus. *Notre Dame, Journal of Formal Logic*, 21(4):685–693, 1980.
20. 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.
21. M. Coppo and A. Ferrari. Type inference, abstract interpretation and strictness analysis. In *A Collection of contributions on honour of Corrado Böhm*, 113–145. Elsevier, 1993.
22. H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.
23. O-J. Dahl and K. Nygaard. SIMULA: an ALGOL-based Simulation Language. *Commun. ACM*, 9(9):671–678, 1966.
24. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings* 9^{th} *ACM Symposium on Principles of Programming Languages*, 207–212, 1982.
25. F. Damiani. Typing local definitions and conditional expressions with rank 2 intersection. In *Proceedings of FOSSACS'00*, volume 1784 of *LNCS*, 82–97, 2000.

26. R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, 198–208. ACM Press, 2000.
27. Ugo de'Liguoro. Subtyping in logical form. *Electr. Notes Theor. Comput. Sci.*, 70(1), 2002.
28. J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Proceedings of 6th International Conference on* Foundations of Software Science and Computational Structures *(FOSSACS'03)*, 2003, 250–266.
29. K. Fisher and F. Honsell and J.C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
30. J.-Y. Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. These D'État, Université Paris VII, 1972.
31. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983
32. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd Edition)*. Prentice Hall, 2005.
33. C. Grothoff. *Expressive Type Systems for Object-oriented Languages*, PhD Thesis, University of California at Los Angeles, 2006.
34. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.
35. T. Jensen. *Abstract Interpretation in Logical Form*. PhD thesis, Imperial College, University of London, November 1992.
36. T. Jensen and F. Spoto. Class Analysis of Object-Oriented Programs through Abstract Interpretation. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, pages 261–275, 2001.
37. A.J. Kfoury, H.G. Mairson, F.A. Turbak, and J.B. Wells. Relating Typability and Expressibility in Finite-Rank Intersection Type Systems. In *Proceedings of ICFP'99, International Conference on Functional Programming*, pages 90–101, 1999.
38. A.J. Kfoury and J. Wells. Principality and decidable type inference for finite-rank intersection types. In *Proceedings of POPL'99:* 26^{th} *ACM Symposium on the Principles of Programming. Languages*, pages 161–174, 1999.
39. M. Kohlhase and F. Pfenning. Unification in a lambda-calculus with intersection types. In *ILPS*, pages 488–505, 1993.
40. D. Leivant. Polymorphic Type Inference. In *Proceedings 10*$^{\text{th}}$ *ACM Symposium on Principles of Programming Languages*, pages 88–98, Austin Texas, 1983.
41. F. Logozzo. Automatic Inference of Class Invariants. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 211–222, January 2004.
42. F. Logozzo. Separate Compositional Analysis of Class-based Object-oriented Languages. In *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST2004), LNCS* 3116, pages 332–346, July 2004.
43. F. Logozzo and A. Cortesi. Abstract Interpretation and Object-oriented Programming: Quo Vadis? In *Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages*, pages 75–84, 2005
44. Microsoft Press. *C# Language Specifications*. Microsoft Press, 2001.
45. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
46. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

47. J.C. Mitchell. Polymorphic Type Inference and Containment. *Information and Computation*, 76:211–249, 1988.
48. J.C. Mitchell. Toward A Typed Foundation for Method Specialization and Inheritance. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–124, 1990.
49. F. Pfenning. Subtyping and intersection types revisited. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany*, page 219, 2007.
50. B.C. Pierce. Intersection Types and Bounded Polymorphism. In *Proceedings of TLCA'93. International Conference on Typed Lambda Calculi and Applications*, Utrecht, the Netherlands, *LNCS* 664, pages 346–360, 1993.
51. B.C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.
52. J. C. Reynolds. Towards a Theory of Type Structures. In B. Robinet, editor, *Proceedings of 'Colloque sur la Programmation'*, Paris, France, *LNCS* 19, pages 408–425, 1974.
53. J. C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pitssburgh, 1988.
54. John C. Reynolds. The coherence of languages with intersection types. In *Proceedings of* Theoretical Aspects of Computer Software *(TACS '91), Sendai, Japan*, pages 675–700, 1991.
55. John C. Reynolds. An intrinsic semantics of intersection types. In *Electronic Proceedings of 3rd International Workshop* Intersection Types and Related Systems *(ITRS'04), Turku, Finland*, pages 269–270, 2000.
56. A. Rountev, A. Milanova, and B. G. Ryder. Points-to Analysis for Java Using Annotated Constraints. In *16th ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA01)*, pages 43–55, November 2001.
57. B. Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991.
58. W. Tait. Intensional interpretation of functionals of finite type I. *The journal of Symbolic Logic 32*, 2, 198–223, 1967.
59. Tachio Terauchi and Alex Aiken. On typability for rank-2 intersection types with polymorphic recursion. In *LICS*, pages 111–122, 2006.
60. A.M. Turing. Computability and Lambda-Definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937.
61. 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.