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

(11th Workshop on Formal Techniques for Java-like Programs (FTfJP'09), 2009)

Steffen van Bakel 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 apply the principles of the intersection type discipline to the study of class-based object oriented programs and; our work follows from a similar approach (in the context of Abadi and Cardelli's ς-object calculus) taken by van Bakel and de'Liguoro. We define an extension of Featherweight Java, *p*FJ, and present a *predicate* system which we show to be sound and expressive. We also show that our system provides a semantic underpinning for the object oriented paradigm by generalising the concept of *approximant* from the Lambda Calculus and demonstrating an approximation result: all expressions to which we can assign a predicate have an approximant that satisfies the same predicate. Crucial to this result is the notion of *predicate language*, which associates a family of predicates with a class.

1 Introduction

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 Lambda Calculus (λ-calculus) [9] and representing objects as records [13, 33, 14, 22]. The seminal work of Abadi and Cardelli [1] constitutes perhaps the most comprehensive formal treatment of object orientation, and introduces the -calculus, which formalises the *object-based* programming paradigm. Similar formal models describing *class-based* languages have been developed as well; notable efforts are Featherweight Java [26] and its successor Middleweight Java [12].

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 be given about their behaviour. 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 [23, 36] as investigated in the early nineties [32, 34, 35, 15], and the *intersection type discipline* (ITD), as first developed in the early 1980s [17, 18, 10, 4] are two good examples of systems which, while undecidable in principle, 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 $\cap$. By introducing this extension a system is obtained that is closed under β-equality: if $B \vdash M : \sigma$ and $M =_\beta N$, then $B \vdash N : \sigma$, making type assignment undecidable. Intersection systems satisfy a number of strong properties that

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

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). The strength of ITD motivated de'Liguoro [19] to apply the principles of intersection types to object oriented programming, in particular to the Varsigma Calculus. Over three papers [5, 6, 7], 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 [7] specifically, to a formal model of *class-based* object oriented programming; the model we use is based on [26]. Having defined the calculus, we will then prove a subject reduction result.

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. To be exact, we show the approximation result: any non-trivial predicate assignment for an expression is also achievable for an approximant of that expression, i.e. a finite rooted segment of its head-normal form. Thus we link semantics and predicates; the head-normal form is assured to exist by the fact that a non-trivial predicate can be assigned. This then can be used as a basis for abstract interpretation; an analysis that is immediately within reach is that of *termination*, as we will show in this paper. This is certainly not the only one however; one could think of dead code analysis, type and effect systems, strictness analysis, etc.

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 [31]; an exception to this is found in [25]. Another observation is that work in this area has been centred around issues of optimisation: [28] presents a *class analysis* of object-oriented programs which may be used to eliminate virtual function calls, *pointer analysis* [37] generalises class analysis and also allows for the detection of null pointer dereferencing, and other analyses [29, 30] have looked at inferring invariants for classes which can be useful in many optimisations such as the removal of checks for array bounds. Termination analysis, missing from this list, is covered by our treatment. Such an analysis has been done on Java bytecode [2], however our system aims at performing such an analysis directly at the level of the object-oriented language rather than its intermediate form.

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 prove an approximation result and characterise head-normalisation and termination. The system, being semi-decidable at best, would need to be limited in expressiveness before it can be used for static analysis. This notwithstanding, the main properties shown in this paper would hold also for such a restriction.

2 Predicate Featherweight Java

In this section we define our extension of Featherweight Java (FJ), which we call Predicate Featherweight Java (or *p*FJ). FJ [26] is a minimal (functional) calculus based on Java [24] which expresses the core features of a class-based object oriented programming language (e.g. inheritance, method invocation and field access, object creation). Its compact nature allows proofs of its properties to be correspondingly succinct. As such, it has proved extremely

$$
\begin{array}{rcl@{\qquad}rcl}
(\texttt{new}\ \mathsf{C}(\vec{\mathsf{e}}_n)).f_i & \to & \mathsf{e}_i, & \mathcal{F}(\mathcal{EC},\mathsf{C}) & = & \vec{f}_n\ \&\ i\in\overline{n};\\
(\texttt{new}\ \mathsf{C}(\vec{\mathsf{e}}_n)).f_i = \mathsf{e}'_i & \to & \texttt{new}\ \mathsf{C}(\mathsf{e}_1,\ldots,\mathsf{e}'_i,\ldots,\mathsf{e}_n), & \mathcal{F}(\mathcal{EC},\mathsf{C}) & = & \vec{f}_n\ \&\ i\in\overline{n};\\
(\texttt{new}\ \mathsf{C}(\vec{\mathsf{e}})).m(\vec{\mathsf{e}'}_n) & \to & \mathsf{e}[\overrightarrow{\mathsf{e}'/x}_n,\texttt{new}\ \mathsf{C}(\vec{\mathsf{e}})/\texttt{this}], & \mathcal{M}b(\mathcal{EC},\mathsf{C},m) & = & (\vec{x}_n,\mathsf{e}).
\end{array}
$$

Figure 1: Reduction rules

popular as a starting point for formally studying extensions to Java [27, 40, 21, 20, 11]. The treatment of FJ and its variants in the literature is very comprehensive, and so here we only define the elements of *p*FJ informally, and discuss its departures from FJ.

Definition 2.1 (*p*FJ Syntax) The syntax of *p*FJ is defined by the following grammar:

$$
\begin{array}{rcl}
\mathsf{cd} & ::= & \texttt{class}\ \mathsf{C}\ \texttt{extends}\ \mathsf{C}'\,\{\,\overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}}\,\}\quad(\mathsf{C}\neq\texttt{Object})\\
\mathsf{md} & ::= & \mathsf{D}\ m(\overrightarrow{\mathsf{C}\ x})\,\{\,\mathsf{e}\,\}\\
\mathsf{fd} & ::= & \mathsf{C}\ f\\
\mathsf{e} & ::= & x \mid \texttt{null} \mid \mathsf{e}.f \mid \mathsf{e}.f = \mathsf{e}' \mid \mathsf{e}.m(\vec{\mathsf{e}}) \mid \texttt{new}\ \mathsf{C}(\vec{\mathsf{e}})\\
\mathcal{EC} & ::= & \overrightarrow{\mathsf{cd}}\\
P & ::= & (\mathcal{EC},\mathsf{e})
\end{array}
$$

The meta-variables C and D range over class names (which, as in FJ, we also use as types); m ranges over method names, f over field identifiers, and x over variable names. The set of class names includes the distinguished name `Object`, and the set of variables includes the distinguished name `this`.

In a similar notation to that of FJ we use $\vec{\mathsf{e}}$ to represent a possibly empty sequence of elements (in this particular case, expressions). When necessary, such a sequence may be subscripted with a meta-variable indicating the number of elements it contains, $\vec{\mathsf{e}}_n$. Notice that elements of a sequence are permitted to be *composite*, as in $\overrightarrow{\mathsf{C}\ x}$. Sequence concatenation is represented by $\vec{\mathsf{e}}\cdot\vec{\mathsf{e}'}$, and denotes the empty sequence. We use $\overline{n}$ for the set $\{1,\ldots,n\}$, where n is a natural number.

Definition 2.2 An *execution context* is a sequence of class declarations, and a program is a pair of an execution context and an expression to be evaluated. Classes contain a list of fields and a list of methods, the (class) types of which must be declared. As in FJ, the superclass must always be explicitly declared even if it is `Object`. Methods may take multiple arguments and method bodies consist of a single expression.

We call $\mathcal{EC}$ an execution context, rather than a class table, in order to highlight its purely syntactic nature (as opposed to some form of lookup).

Notice that *p*FJ does *not* explicitly include constructors, as FJ does. We have chosen to elide this feature since in FJ it is merely 'syntactic sugar': constructor methods are never *run*, in the same sense that other methods are invoked, and were included to ensure that all valid FJ programs are also valid Java programs. In *p*FJ, we make object constructors *implicit* by requiring (in the type rule for the `new` keyword) that the types of the expressions that are to be assigned as field values match the types for the fields as defined by the class of the object being created. We also omit the `return` keyword in method bodies for the same reason. We feel that this simplifies the calculus without diminishing its relevance in any way.

An important difference between *p*FJ and FJ is that we omit cast expressions in *p*FJ, which were included in FJ in order to support the compilation of Featherweight GJ programs to FJ [26, §3]. Since that is not an objective of our work, and (more importantly) the presence of

downcasts makes the system unsound in the sense that well-typed expressions can reduce to expressions containing meaningless (or 'stupid') casts, they are omitted. Upcasts are replaced by subsumption rules in the type system. In *p*FJ we also include syntax to represent the `null` value and a field assignment operation. One of the objectives of our research is to lay a foundation for the treatment of a *stateful* model of object oriented programs, of which these two elements are quintessential components. We therefore feel that it behooves us to incorporate them into the model at the earliest opportunity. Indeed, even at the functional level, we find that their inclusion has some interesting (and non-trivial) consequences: our predicate system can be made expressive enough to catch 'null pointer dereferences', and field assignment has important implications for the definition of predicate languages in a complete system.

Definition 2.3 We use the (syntactic) execution context to define a family of standard lookup functions:

i) $\mathcal{F}(\mathcal{EC},\mathrm{C}) = \vec{f}$ returns a sequence of the fields defined (and inherited by) class C;
ii) $\mathcal{M}b(\mathcal{EC},\mathrm{C},m) = (\vec{x},\mathtt{e})$ returns the body e of the method m in class C, along with a sequence containing the names of its formal parameters;
iii) $\mathcal{FT}(\mathcal{EC},\mathrm{C},f) = \mathrm{D}$ returns the type of field f in class C;
iv) $\mathcal{MT}(\mathcal{EC},\mathrm{C},m) = \vec{\mathrm{C}} \to \mathrm{D}$ returns the signature of method m in class C.

We explicitly define the lookup functions such that the class `Object` is empty (i.e. contains no fields or methods). This is safe since the grammar of *p*FJ precludes the existence of a user-defined class called `Object`. An execution context induces a standard subtype relation $<:$ defined as the transitive closure of the of class extension.

A number of notions and concepts are defined that strongly depend on the current execution context (like reduction, type assignment, and predicate assignment), and which, therefore, should all be subscripted with its name; but since this context is not changed by execution, we will not do so. As usual, we impose some well-formedness conditions on execution contexts (e.g. all classes must be uniquely named, and the class hierarhy must be acyclic).

i) all classes are uniquely named;
ii) the class hierarchy is *acyclic*;
iii) no class declares a field which it also inherits;
iv) if a class declares a method which it also inherits, then the declared signature must match that of the inherited method;
v) the variable `this` is not used as a formal parameter to any method;
vi) the types declared for fields and in method signatures must correspond to valid classes, as must all classes that are inherited from.

Notice that in *well-formed execution contexts* we explicitly forbid the redeclaration of an inherited field, but we allow methods declared higher up in the inheritance hierarchy to be overridden (redefined) in a subclass, subject to the condition that the type signature is identical. Such behaviour is a common (perhaps even integral) aspect of the object oriented paradigm and is also present in FJ.

2.1 Reduction

We retain the permissive reduction of FJ (rather than restrict it to a call-by-value semantics as in other extensions, e.g. [11]) and extend it to handle field assignment. As in FJ, we use $\mathtt{e}[\overrightarrow{\mathtt{e}'/x_n}]$ to denote the expression obtained by replacing any occurrences of the variables $x_1, \ldots, x_n$ in

e by the expressions $e_1, \ldots, e_n$ respectively. Formally, a reduction relation $\rightarrow_{\mathcal{EC}}$ is induced for each execution context; however, as mentioned previously, from now on we will assume a fixed execution context.

Definition 2.4 (*p*FJ REDUCTION) The one-step reduction relation is defined as the contextual closure of the rules given in Figure 1.

2.2 Type System

The types of *p*FJ are the same as those of FJ; that is, they are induced by the set of classes defined in the execution context, augmented with `Object`. We modify the type system of FJ to handle our extra syntax in an obvious way: we introduce extra rules to allow `null` to be assigned any valid type, and ensure that the r-value in a field assignment expression has the expected type. We also introduce a separate subsumption rule.

Definition 2.5 *i*) If a class C is defined in an execution context $\mathcal{EC}$, then we say it is *valid in* $\mathcal{EC}$; `Object` is valid in any execution context.

ii) A type environment is a set of statements of the form x:C, which is *well formed* when each statement refers to a uniquely named variable x and a valid type C.

iii) The typing judgement of *p*FJ is written as $\Gamma \vdash \mathsf{e}$:C – where Γ and $\mathcal{EC}$ are well formed – which reads: e has type C in the type environment Γ. The rules of the type assignment system are given by:

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

$$[\textsc{T-NULL}]: \frac{}{\Gamma \vdash \texttt{null}\text{:C}} \; (\text{C valid in } \mathcal{EC}) \qquad [\textsc{T-SUB}]: \frac{\Gamma \vdash \mathsf{e}\text{:C}'}{\Gamma \vdash \mathsf{e}\text{:C}} \; (\text{C}' <: \text{C})$$

$$[\textsc{T-INVK}]: \frac{\Gamma \vdash \mathsf{e}\text{:C} \quad \Gamma \vdash \mathsf{e}_i\text{:C}_i \; (\forall i \in \overline{n})}{\Gamma \vdash \mathsf{e}.m(\vec{\mathsf{e}}_n)\text{:D}} \; (\mathcal{MT}(\mathcal{EC},\text{C},m) = \vec{\text{C}}_n \rightarrow \text{D}) \qquad [\textsc{T-VAR}]: \frac{}{\Gamma \vdash x\text{:C}} \; (x\text{:C} \in \Gamma)$$

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

iv) An execution context is *type consistent* if and only if the execution context is well formed and the body of each method can be assigned its declared return type under the type assumptions given for its parameters in the method signature.

As for FJ, we can show a soundness result for *p*FJ:

Theorem 2.6 *For type consistent execution contexts if* $\Gamma \vdash \mathsf{e}$:C *and* $\mathsf{e} \rightarrow \mathsf{e}'$ *then* $\Gamma \vdash \mathsf{e}'$:C

3 The Predicate System

We now come to describe the first contribution of our work: the predicate system. Our system aims to provide an analysis which is more expressive than the simple type system of FJ: rather than simply guaranteeing global properties of programs, we wish our predicate types to be semantic in nature, and capture runtime properties. We consider the behaviour of an expression (or rather, the object to which the expression evaluates) in terms of the operations that we may perform on it, i.e. accessing a field or invoking a method. We follow in the tradition of intersection types, originally defined as sequences [16], however, by treating our predicates as such: a predicate is a sequence of (potentially incomparable) behaviours, from

$$[\textsc{p-null}]: \frac{}{\Pi \vdash \texttt{null}:\mathrm{C}:} \ (\mathrm{C} \text{ valid in } \mathcal{X}) \qquad [\textsc{p-var}]: \frac{}{\Pi \vdash x:\mathrm{C}:\nu} \ (x:\mathrm{C}:\nu \in \Pi)$$

$$[\textsc{p-newO}]: \frac{\Pi \vdash \texttt{new}\ \mathrm{C}(\vec{\mathsf{e}}):\mathrm{C}}{\Pi \vdash \texttt{new}\ \mathrm{C}(\vec{\mathsf{e}}):\mathrm{C}:\langle\rangle} \qquad [\textsc{p-fld}]: \frac{\Pi \vdash \mathsf{e}:\mathrm{D}:\langle f:\nu\rangle}{\Pi \vdash \mathsf{e}.f:\mathrm{C}:\nu} \ (\mathcal{FT}(\mathcal{X},\mathrm{D},f)=\mathrm{C})$$

$$[\textsc{p-subT}]: \frac{\Pi \vdash \mathsf{e}:\mathrm{C}':\nu}{\Pi \vdash \mathsf{e}:\mathrm{C}:\nu} \ (\mathrm{C}' <: \mathrm{C} \ \&\ \nu \in \mathcal{L}(\mathrm{C})) \qquad [\textsc{p-top}]: \frac{\Pi \vdash \mathsf{e}:\mathrm{C}}{\Pi \vdash \mathsf{e}:\mathrm{C}:\top}$$

$$[\textsc{p-join}]: \frac{\Pi \vdash \mathsf{e}:\mathrm{C}:\sigma_i \quad (\forall i \in \overline{n})}{\Pi \vdash \mathsf{e}:\mathrm{C}:\sqcup \vec{\sigma}_n} \ (n>0) \qquad [\textsc{p-seq}]: \frac{\Pi \vdash \mathsf{e}:\mathrm{C}:\sigma'}{\Pi \vdash \mathsf{e}:\mathrm{C}:\sigma} \ (\sigma' \trianglelefteq \sigma \ \&\ \sigma \in \mathcal{L}(\mathrm{C}))$$

$$[\textsc{p-ass}_1]: \frac{\Pi \vdash \mathsf{e}:\mathrm{C}:\sigma \quad \Pi \vdash \mathsf{e}':\mathrm{D}:\nu}{\Pi \vdash \mathsf{e}.f=\mathsf{e}':\mathrm{C}:\langle f:\nu\rangle} \ (\mathcal{FT}(\mathcal{X},\mathrm{C},f)=\mathrm{D})$$

$$[\textsc{p-ass}_2]: \frac{\Pi \vdash \mathsf{e}:\mathrm{C}:\langle \overrightarrow{\ell:\tau}_n\rangle \quad \Pi \vdash \mathsf{e}':\mathrm{D}}{\Pi \vdash \mathsf{e}.f=\mathsf{e}':\mathrm{C}:\langle \overrightarrow{\ell:\tau}_n\rangle} \ (f \notin \vec{\ell}_n \ \&\ \mathcal{FT}(\mathcal{X},\mathrm{C},f)=\mathrm{D})$$

$$[\textsc{p-invk}]: \frac{\Pi \vdash \mathsf{e}:\mathrm{D}:\langle m:\psi :: \vec{\phi}_n \to \nu\rangle \quad \Pi \vdash \mathsf{e}:\mathrm{D}:\psi \quad \Pi \vdash \mathsf{e}_i:\mathrm{C}_i:\phi_i \ \ (\forall i \in \overline{n})}{\Pi \vdash \mathsf{e}.m(\vec{\mathsf{e}}_n):\mathrm{C}:\nu} \ (\mathcal{MT}(\mathcal{X},\mathrm{D},m)=\vec{\mathrm{C}}_n \to \mathrm{C})$$

$$[\textsc{p-newF}]: \frac{\Pi \vdash \mathsf{e}_j:\mathrm{C}_j:\nu \quad \Pi \vdash \mathsf{e}_i:\mathrm{C}_i \quad (\forall i \in \overline{n}\ [\ i \neq j\ ])}{\Pi \vdash \texttt{new}\ \mathrm{C}(\vec{\mathsf{e}}_n):\mathrm{C}:\langle f_j:\nu\rangle} \ (\ \mathcal{F}(\mathcal{X},\mathrm{C})=\vec{f}_n \ \&\ j \in \overline{n} \ \&\ \forall\, i \in \overline{n}\ [\ \mathcal{FT}(\mathcal{X},\mathrm{C},f_i)=\mathrm{C}_i\ ]\ )$$

$$[\textsc{p-newM}]: \frac{\Pi \vdash \texttt{new}\ \mathrm{C}(\vec{\mathsf{e}}):\mathrm{C} \quad \Pi' \vdash \mathsf{e}_0:\mathrm{D}:\nu}{\Pi \vdash \texttt{new}\ \mathrm{C}(\vec{\mathsf{e}}):\mathrm{C}:\langle m:\psi :: \vec{\phi}_n \to \nu\rangle}$$
$$(\mathcal{MT}(\mathcal{X},\mathrm{C},m)=\vec{\mathrm{C}}_n \to \mathrm{D} \ \&\ \mathcal{Mb}(\mathcal{X},\mathrm{C},m)=(\vec{x}_n,\mathsf{e}_0) \ \&\ \Pi'=\{\overrightarrow{x:\mathrm{C}:\phi}_n, \texttt{this}:\mathrm{C}:\psi\})$$

Figure 2: Predicate Assignment Rules

which any specific one can be selected for an expression as demanded by to the context in which it appears. We also incorporate the late typing of self, another important feature found in other intersection type systems for object calculi [8, 5]. This allows for a greater flexibility in the system, permitting us to update an object prior to invoking a method on it.

We begin by defining our predicate types.

Definition 3.1 (Predicates) Predicates are defined by the following grammar:

$$\begin{array}{lll} \textit{predicates}: & \phi & ::= \ \top \mid \nu \\ \textit{normal predicates}: & \nu & ::= \ \mid \sigma \\ \textit{object predicates}: & \sigma & ::= \ \langle \overrightarrow{\ell:\tau}\rangle \\ \textit{member predicates}: & \tau & ::= \ \nu \mid \psi :: \vec{\phi} \to \nu \end{array}$$

where the meta-variable ℓ ranges over the set of both field identifiers and method names.

Object predicates thus comprise a sequence of statements describing the behaviour of an object. Each statement associates a certain behaviour (described by the member predicate τ) with the result of accessing the field or invoking the method labelled ℓ. In the case of methods, the predicate additionally indicates the *required* behaviour of the receiver (ψ) and the arguments ($\vec{\phi}$). By combining the predicate (denoting a null value) with the object predicates we obtain the set of *normal* predicates, so called because they can be assigned to expressions which evaluate to safe normal forms[2]. The predicate constant $\top$ (top) is a standard feature taken from the intersection type discipline, and has the role of covering expressions which do not terminate or, more generally, the result of which bears no relevance to the running of the

[2] The normal forms are safe in the sense that they do not contain null pointer dereferences.

$$
\begin{array}{rl}
p = \top,, \langle\,\rangle \Rightarrow & \mathit{Comp}(\Pi, \mathsf{e}{:}\mathrm{C}, p) \;\Leftrightarrow\; \mathit{Appr}_{\mathscr{EC}}(\Pi, \mathsf{e}{:}\mathrm{C}, p) \\
\Pi \vdash \mathsf{e}{:}\mathrm{C} \;\&\; \mathcal{FT}(\mathscr{EC}, \mathrm{C}, f) = \mathrm{D} \Rightarrow & (\mathit{Comp}(\Pi, \mathsf{e}{:}\mathrm{C}, \langle f : \nu \rangle) \;\Leftrightarrow\; \mathit{Comp}(\Pi, \mathsf{e}.f{:}\mathrm{D}, \nu)) \\
\Pi \vdash \mathsf{e}{:}\mathrm{C} \;\&\; \mathcal{MT}(\mathscr{EC}, \mathrm{C}, m) = \vec{\mathrm{C}}_n \to \mathrm{D} \Rightarrow & (\mathit{Comp}(\Pi, \mathsf{e}{:}\mathrm{C}, \langle m : \psi :: \vec{\phi}_n \to \nu \rangle) \;\Leftrightarrow \\
& \quad (\mathit{Comp}(\Pi, \mathsf{e}{:}\mathrm{C}, \psi) \;\&\; \forall i \in \overline{n}\; [\; \mathit{Comp}(\Pi, \mathsf{e}_i{:}\mathrm{C}_i, \phi_i)\;] \;\Rightarrow\; \mathit{Comp}(\Pi, \mathsf{e}.m(\vec{\mathsf{e}}_n){:}\mathrm{D}, \nu))) \\
& \quad \forall i \in \overline{n}\; [\; \mathit{Comp}(\Pi, \mathsf{e}{:}\mathrm{C}, \sigma_i)\;] \;\Leftrightarrow\; \mathit{Comp}(\Pi, \mathsf{e}{:}\mathrm{C}, \sqcup\, \vec{\sigma}_n)\;\; (n > 0)
\end{array}
$$

Figure 3: Computability predicate

program in that it does not influence the final outcome. Notice that intersections are implicitly present in the object predicates, since there is no restriction in place on the labels used: a label can occur more than once. This corresponds to the approach of the strict intersection system [3].

We now define a subpredicate relation and an operation which combines (object) predicates together. At the heart of intersection type assignment lies the ability to introduce an intersection of types and select a single type from an intersection. In our system the join operation facilitates the former (intersection introduction), and the subpredicate relation allows us to perform intersection elimination.

Definition 3.2 (Subpredicate Relation) The relation $\trianglelefteq$ is defined as the least pre-order on predicates such that:

$$
\begin{array}{rcll}
 & \trianglelefteq & \top & \forall i \in \overline{n}\,[\,\langle \overrightarrow{\ell : \tau}_n \rangle \;\trianglelefteq\; \langle \ell_i{:}\tau_i \rangle\,] \\
\langle\,\rangle & \trianglelefteq & \top & \forall i \in \overline{n}\,[\,\sigma \trianglelefteq \langle \ell_i{:}\tau_i \rangle\,] \;\Rightarrow\; \sigma \trianglelefteq \langle \overrightarrow{\ell : \tau}_n \rangle \quad (n \geq 0)
\end{array}
$$

Again, this corresponds to the type inclusion relation for strict types.

Definition 3.3 (Predicate Join) The *join* operation is defined on object predicates as follows:

$$\langle \overrightarrow{\ell{:}\tau} \rangle \sqcup \langle \overrightarrow{\ell'{:}\tau'} \rangle = \langle \overrightarrow{\ell{:}\tau} \cdot \overrightarrow{\ell'{:}\tau'} \rangle$$

We generalise the join operation to sequences of object predicates as follows:

$$\sqcup \;=\; \langle\,\rangle \qquad \sqcup\, \sigma \cdot \vec{\sigma} \;=\; \sigma \sqcup (\sqcup\, \vec{\sigma})$$

Since the motivating idea behind predicates is to make a statement on the execution of an expression, we define the notion of a predicate language which allows our system to be truly predictive. For example, by defining this notion, we can show that if we derive the predicate $\langle f : \nu \rangle$ for a typed expression $\mathsf{e}{:}\mathrm{C}$, then the field f will be visible in the class C. Moreover, it will be safe to access the field in e, and the result will satisfy the predicate ν.

Definition 3.4 (Predicate Language) $\mathcal{L}(\mathrm{C})$, the *language* of class C is the smallest set of predicates satisfying the following conditions:

i) $\top \in \mathcal{L}(\mathrm{C})$, $\in \mathcal{L}(\mathrm{C})$ and $\langle\,\rangle \in \mathcal{L}(\mathrm{C})$.

ii) $\mathcal{FT}(\mathscr{EC}, \mathrm{C}, f) = \mathrm{D} \Leftrightarrow (\nu \in \mathcal{L}(\mathrm{D}) \Leftrightarrow \langle f{:}\nu \rangle \in \mathcal{L}(\mathrm{C}))$.

iii) $\mathcal{MT}(\mathscr{EC}, \mathrm{C}, m) = \vec{\mathrm{C}}_n \to \mathrm{D} \Leftrightarrow$
$(\psi \in \mathcal{L}(\mathrm{C}) \;\&\; \forall i \in \overline{n}\,[\,\phi_i \in \mathcal{L}(\mathrm{C}_i)\,] \;\&\; \nu \in \mathcal{L}(\mathrm{D}) \Leftrightarrow$
$\langle m : \psi :: \vec{\phi}_n \to \nu \rangle \in \mathcal{L}(\mathrm{C}))$.

iv) $\forall i \in \overline{n}\,[\sigma_i \in \mathcal{L}(\mathrm{C})] \Leftrightarrow \sqcup\, \vec{\sigma}_n \in \mathcal{L}(\mathrm{C})$.

This notion of language plays a crucial role in the approximation result that is presented in the next section.

Definition 3.5 The rules for our predicate assignment system are given in Figure 2. A predicate environment Π, which is a set of statements x:C:ϕ, is well formed if each statement refers to a unique variable x, a valid type C, and a predicate $\phi \in \mathcal{L}(\mathrm{C})$. The judgement $\Pi \vdash \mathsf{e}{:}\mathrm{C} : \phi$ – where again Π and $\mathcal{EC}$ are well formed – asserts that the expression e of type C can be assigned the predicate ϕ using Π.

Some rules are premised by type assignment judgements, which we write using predicate environments instead of type environments ($\Pi \vdash \mathsf{e}{:}\mathrm{C}$). Notice that this is more than a simple notational convenience: formally this is a sound extension since each type environment corresponds to a predicate environment in which the predicate information has been discarded.

We can see the predicate system as a Hoare-style system of pre- and post-conditions. For example, the rule (P-FLD) expresses that if the expression e satisfies the predicate $\langle f : \nu \rangle$, then accessing the field f will satisfy ν, giving an annotation like

$$\begin{array}{ll} :: & \texttt{pre: e satisfies } \langle f : \nu \rangle \\ \mathsf{e}.f & \\ :: & \texttt{post: } \nu \end{array}$$

As a final comment, we return to the issue of late self typing, mentioned earlier in this section. Notice that a method predicate $\langle m : \psi :: \vec{\phi} \to \nu \rangle$ is derived only for new object expressions[3] using the (P-NEWM) rule, and that no information about this object (save for its type, which allows us to look up the correct method body) is used to derive the self predicate ψ. It is only at *the point of invocation* that we check the receiver to ensure it satisfies ψ. This approach differs from the type systems of [1] for the -calculus, where the self reference in the body of a method may only be given a type reflecting the *current* state of the receiver, even though it may be updated later.

We now present the main results of the predicate system.

Theorem 3.6 *i)* $\exists\, \phi\, [\, \Pi \vdash \mathsf{e}{:}\mathrm{C} : \phi\,] \Leftrightarrow \Pi \vdash \mathsf{e}{:}\mathrm{C}$.
ii) $\Pi \vdash \mathsf{e}{:}\mathrm{C} : \phi \Rightarrow \phi \in \mathcal{L}(\mathrm{C})$.
iii) For type consistent execution contexts if $\Pi \vdash \mathsf{e}{:}\mathrm{C} : \phi$ *and* $\mathsf{e} \to \mathsf{e}'$ *then* $\Pi \vdash \mathsf{e}'{:}\mathrm{C} : \phi$.

4 Approximants for pFJ

In this section, we derive an approximation result which can be used as a basis for semantics-based abstract interpretation, or, more directly, a termination analysis of pFJ. It also opens the way forward for giving a denotational semantics to our calculus.

The notion of approximant was first introduced for the λ-calculus by C. Wadsworth [39]. Intuitively, an approximant can be seen as a 'snapshot' of a computation, constructed by covering places where computation (reduction) may still take place with the element Ω[4].

Definition 4.1 We define *approximate* pFJ expressions by the following grammar:

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

[3] This is not strictly true, since we might also derive a method predicate for a variable when it is mentioned in the environment.

[4] Ω is the symbol originally used in [39]; more common now is to, as [9], use the symbol $\bot$; since this could be confused with our predicate $\top$, we have opted for the old notation.

By extending the notion of reduction so that any field access, field assignment or method invocation on Ω itself reduces to the expression Ω, we can also define the notion of approximate *normal forms*.

Definition 4.2 Approximate normal forms are defined by the following grammar:

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

We extend the type and predicate assignment relations to operate over approximate expressions. We add a type assignment rule permitting Ω to have any valid type, however we do not modify the predicate assignment rules. In particular, this means that Ω *must* be assigned the predicate $\top$.

To formalise the notion of *snapshot*, we define an ordering on approximate expressions:

Definition 4.3 The *direct approximation* relation $\sqsubseteq$ over approximate expressions is defined as the smallest pre-order satisfying:

$$\begin{array}{rcl} \Omega & \sqsubseteq & \mathsf{e} \\ \mathsf{e} \sqsubseteq \mathsf{e}' & \Rightarrow & \mathsf{e}.f \sqsubseteq \mathsf{e}'.f \\ \mathsf{e}_1 \sqsubseteq \mathsf{e}_1' \ \&\ \mathsf{e}_2 \sqsubseteq \mathsf{e}_2' & \Rightarrow & \mathsf{e}_1.f = \mathsf{e}_2 \sqsubseteq \mathsf{e}_1'.f = \mathsf{e}_2' \\ \mathsf{e} \sqsubseteq \mathsf{e}' \ \&\ \mathsf{e}_i \sqsubseteq \mathsf{e}_i' \text{ for all } i \in \overline{n} & \Rightarrow & \mathsf{e}.m(\vec{\mathsf{e}}_n) \sqsubseteq \mathsf{e}'.m(\vec{\mathsf{e}'}_n) \\ \mathsf{e}_i \sqsubseteq \mathsf{e}_i' \text{ for all } i \in \overline{n} & \Rightarrow & \texttt{new}\ \mathrm{C}(\vec{\mathsf{e}}_n) \sqsubseteq \texttt{new}\ \mathrm{C}(\vec{\mathsf{e}'}_n) \end{array}$$

An *approximant* of an expression e is an approximate normal form A which directly approximates some expression e' to which e reduces, except for occurrence of Ω in A (so $\mathsf{A} \sqsubseteq \mathsf{e}'$). We write $\mathcal{A}(\mathsf{e})$ to denote the set of all the approximants of e.

The following result gives an approximation semantics to p_{FJ}, in which we interpret an expression by its set of approximants, $[\![\mathsf{e}]\!] = \mathcal{A}(\mathsf{e})$.

Lemma 4.4 $\mathsf{e} \rightarrow^* \mathsf{e}' \Rightarrow \mathcal{A}(\mathsf{e}) = \mathcal{A}(\mathsf{e}')$

As a shorthand notation, we define an *approximation* predicate:

Definition 4.5 $Appr_{\mathscr{C}}(\Pi, \mathsf{e}{:}\mathrm{C}, \phi) \Leftrightarrow \Pi \vdash \mathsf{e}{:}\mathrm{C}\ \&\ \exists \mathsf{A} \in \mathcal{A}(\mathsf{e})\ [\ \Pi \vdash \mathsf{A}{:}\mathrm{C} : \phi\ ]$.

The approximation result is the following: any expression to which a predicate can be assigned has an approximant with that same predicate. We follow Tait's proof method [38] involving a *computability* predicate. Space restrictions do not allow us to present the proofs in detail.

Definition 4.6 (Computability Predicate) The computability predicate is defined inductively over predicates as in Figure 3.

A key step in the proof is to show that computability implies approximation:

Lemma 4.7 *i)* $Comp(\Pi, \mathsf{e}{:}\mathrm{C}, \phi) \Rightarrow Appr_{\mathscr{C}}(\Pi, \mathsf{e}{:}\mathrm{C}, \phi)$.
ii) $\Pi \vdash x{:}\mathrm{C} : \phi \Rightarrow Comp(\Pi, x{:}\mathrm{C}, \phi)$.

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 4.8 (Replacement Lemma) *If* $\Pi \vdash \mathsf{e}{:}\mathsf{C} : \phi$*, and there exists* Π', $\overrightarrow{\mathsf{e}'}_n$ *such that for all* $x_i{:}\mathsf{C}_i{:}\phi_i \in \Pi$ *we have that* $Comp(\Pi', \mathsf{e}_i{:}\mathsf{C}_i, \phi_i)$*, then* $Comp(\Pi', \mathsf{e}[\overrightarrow{x/\mathsf{e}}_n]{:}\mathsf{C}, \phi)$.

Given that all variables are computable of predicates which are assignable to them (Lemma 4.7), we can simply replace all the variables in an expression by themselves, and so a corollary of the replacement lemma is that if an expression can be assigned a predicate then it is also computable of that predicate.

Corollary 4.9 $\Pi \vdash \mathsf{e}{:}\mathsf{C} : \phi \Rightarrow Comp(\Pi, \mathsf{e}{:}\mathsf{C}, \phi)$.

Combining this with Lemma 4.7 allows us to derive our approximation result:

Theorem 4.10 *If* $\Pi \vdash \mathsf{e}{:}\mathsf{C} : \phi$ *then there exists* $\mathsf{A} \in \mathcal{A}(\mathsf{e})$ *such that* $\Pi \vdash \mathsf{A}{:}\mathsf{C} : \phi$.

While the approximation result shown above is significant in its own right, perhaps of more interest is that it facilitates a *termination analysis* of *p*FJ. We can show that all expressions to which we can assign a *normal* predicate (i.e. not $\top$) have a *head-normal form*, that is they will reduce to either the null value or an object[5].

Definition 4.11 (Head normal forms) Head normal forms for *p*FJ are defined by the following grammar:

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

Theorem 4.12 (Termination) *If* $\Pi \vdash \mathsf{e}{:}\mathsf{C} : v$ *then there exists* H *such that* $\mathsf{e} \rightarrow^* \mathsf{H}$.

To illustrate this result, consider the following program:

Example 4.13 Take the environment

```
class C extends Object {
  C f
  C m() { this.f }
}
```

Notice that the expression `(new C(null)).m()` has the approximant `null` (which is also its normal form). We can easily derive $\emptyset \vdash \texttt{null}{:}\texttt{C}:$ using the (P-NULL) rule. The following derivation shows that we can also assign this predicate to the original expression:

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{this}{:}\texttt{C}{:}\langle \texttt{f}{:}\rangle\} \vdash \texttt{this}{:}\texttt{C}{:}\langle \texttt{f}{:}\rangle}}{\{\texttt{this}{:}\texttt{C}{:}\langle \texttt{f}{:}\rangle\} \vdash \texttt{this.f}{:}\texttt{C}:} \quad \dfrac{}{\emptyset \vdash \texttt{null}{:}\texttt{C}:}}{\dfrac{\emptyset \vdash \texttt{new C(null)}{:}\texttt{C}{:}\langle \texttt{m}{:}\langle \texttt{f}{:}\rangle ::\rightarrow\rangle \quad \emptyset \vdash \texttt{new C(null)}{:}\texttt{C}{:}\langle \texttt{f}{:}\rangle}{\emptyset \vdash \texttt{(new C(null)).m()}{:}\texttt{C}:}}$$

5 What About Completeness?

While the system we have presented in this paper is sound (exhibits subject reduction), it is not completely expressive since predicable approximants may exist for an expression to which we

[5] This holds for expressions typed in an empty environment (closed expressions). In general, a head normal form may also comprise a sequence of field accesses, assignment and method invocations on variables.

cannot assign those same predicates. This is a consequence of the fact that our system does not have a subject expansion property (as other intersection type assignment systems do). While this may easily be achieved by discarding our notion of predicate language, doing so would destroy the semantic underpinning of our system (i.e. the approximation result). The challenge, therefore, is to construct a system with both properties. While we do not offer a comprehensive solution here, we will discuss the underlying reasons for the failure of subject expansion in the presence of predicate languages as we have defined them, and discuss, at an abstract level, the steps that will be required.

This issue goes right to the heart of the object oriented paradigm since the failure of subject expansion in our system lies in the *dynamic dispatch* feature of OO.

Example 5.1 Take the program

```
class Sub extends Object {
  A upcast(A x) { x }
}
class A extends Object {
  A m() { this }
}
class B extends A {
  A f
  A m() { this.f }
}
```

and the run

```
   (new Sub()).upcast(new B(new A())).m()   (1)
→  (new B(new A())).m()                     (2)
→  (new B(new A())).f                       (3)
→   new A()                                 (4)
```

We begin by invoking the method `m` on the receiving expression `(new Sub()).upcast(new B(new A()))`. By looking at the execution context, we see that the `upcast` method returns a result of type `A`. However, at runtime, the result is actually an object of type `B`, namely `new B(new A())`. Thus, the method body that will be executed when `m()` is invoked will the one found in class `B`. So, are we able to derive a predicate for `(new Sub()).upcast(new B(new A()))` that will allow the call to `m()` be typed?

In order to do this, we must find a predicate (assignable to the object `new Sub()`) for the `upcast()` method such that the result is the predicate mentioning `m` that we desire. This will be of the form $\langle \texttt{upcast}{:}\psi :: \nu \to \nu \rangle$ since the `upcast` method simply returns its argument. However, in addition to ν mentioning `m`, it *must* be the case that both $\nu \in \mathcal{L}(\texttt{A})$ *and* $\Pi \vdash \texttt{new B(new A())}{:}\texttt{A}{:}\nu$. Here we come to heart of the matter: in order to derive a predicate describing the method `m` which we can assign to `new B(new A())`, we must look at the body of `m` in `B`. This method body refers to the field `f` in the receiver (`this`), and thus any predicate which we derive must also mention `f`. However, since `f` is not visible in the type `A` any such predicate will not be in the language of `A`. We find that there is no predicate ν which satisfies the necessary criteria and so we will not be able to assign a (non-trivial) predicate to expression (1) even though we can do so for its normal form, expression (4): e.g. $\emptyset \vdash \texttt{new A()}{:}\texttt{A}{:}\langle\rangle$ by using rule (P-NEWO).

Given that predicate languages are an essential element for the predictive abilities that we desire, the solution to the expansion problem will have to consist in modifying the definition

of predicate languages to make them more permissive. In the example discussed above, we required a predicate to mention members which were not visible in the class of the language to which it belonged. Clearly, we must be able to allow predicates to contain such information, but only in the cases where it is necessary for expansion to hold since we still require that predicate languages make a statement about what is visible in a class and what is not.

6 Conclusions and Future Work

We have presented a predicate (type) system for pFJ, a variant of FJ, and shown that our predicates describe semantic properties of expressions. Our system thus has more expressive power than traditional type systems for Java. We see our results as important initial steps along the road to building not only semantic models for object oriented programming, but also practical analytic systems. A key development towards this aim will be to extend our system to a *stateful* programming model, akin to Middleweight Java [12]. Another objective of immediate concern to us is that of addressing the issues discussed in §5 and achieving subject expansion.

References

[1] M. Abadi and L. Cardelli. *A Theory Of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS*, pp. 2–18, 2008.

[3] S. van Bakel. Complete restrictions of the Intersection Type Discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.

[4] S. van Bakel. Intersection Type Assignment Systems. *Theoretical Computer Science*, 151(2):385–435, 1995.

[5] S. van Bakel and U. de'Liguoro. Logical Semantics for the First Order Sigma Calculus. In *ICTCS'03*, LNCS 2841, pp. 202–215. Springer-Verlag, 2003.

[6] S. van Bakel and U. de'Liguoro. Logical Semantics for $\mathsf{FOb}_{1<:\mu}$. In *ICTCS'05*, LNCS 3701, pp. 66–80. Springer-Verlag, 2005.

[7] S. van Bakel and U. de'Liguoro. Logical Equivalence for Subtyping Object and Recursive Types. *Theory of Computing Systems*, 42(3):306–348, 2008.

[8] F. Barbanera and U. de'Liguoro. Type Assignment for Mobile Objects. *ENTCS*, 104:25–38, 2004.

[9] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, 1981.

[10] 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.

[11] L. Bettini, S. Capecchi, and B. Venneri. Featherweight Java with Multi-Methods. In *PPPJ, volume 272 of ACM International Conference Proceeding Series*, pp. 83–92. ACM, 2007.

[12] 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 Laboratory, April 2003.

[13] L. Cardelli. A Semantics of Multiple Inheritance. In *Proc. of the international symposium on Semantics of data types*, pp. 51–67, 1984, Springer-Verlag.

[14] L. Cardelli and J. C. Mitchell. Operations on Records. In *Proceedings of the fifth international conference on Mathematical foundations of programming semantics*, pp. 22–52, Springer-Verlag.

[15] G. Castagna and B. Pierce. Decidable bounded quantification. In *POPL'94*, pp. 151–162, 1994.

[16] M. Coppo and M. Dezani-Ciancaglini. A New Type Assignment for Lambda-Terms. *Archive für Mathematischer Logic und Grundlagenforschung*, 19:139–156, 1978.

[17] 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.

[18] 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.

[19] Ugo de'Liguoro. Subtyping in Logical Form. *ENTCS*, 70(1), 2002.
[20] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *Proceedings of ECOOP'06, LNCS*, pp. 328–352. Springer-Verlag, 2006.
[21] E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In *POPL06*, pp. 270–282. ACM Press, 2006.
[22] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic J. of Computing*, 1(1):3–37, 1994.
[23] J. Y. Girard. *Interprtation fonctionnelle et limination des coupures de l'arithmtique d'ordre suprieur*. PhD thesis, Universit Paris VII, 1972.
[24] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd Edition)*. Prentice Hall, 2005.
[25] C. Grothoff. *Expressive Type Systems for Object-oriented Languages*. PhD thesis, UCLA, 2006.
[26] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM Transactions on Programming Languages and Systems* (TOPLAS), 23(3), May 2001.
[27] A. Igarashi and B. C. Pierce. On Inner Classes. In *Information and Computation*, 2000.
[28] T. P. Jensen and F. Spoto. Class analysis of object-oriented programs through abstract interpretation. In *FoSSaCS*, pp. 261–275, 2001.
[29] F. Logozzo. Automatic inference of class invariants. In *VMCAI*, pp. 211–222, 2004.
[30] F. Logozzo. Separate compositional analysis of class-based object-oriented languages. In *AMAST*, pp. 334–348, 2004.
[31] F. Logozzo and A. Cortesi. Abstract interpretation and object-oriented programming: Quo vadis? *Electr. Notes Theor. Comput. Sci.*, 131:75–84, 2005.
[32] J. C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2/3):211–249, 1988.
[33] J. C. Mitchell. Toward A Typed Foundation for Method Specialization and Inheritance. In *POPL '90*, pp. 109–124, 1990. ACM.
[34] B. C. Pierce. Intersection types and bounded polymorphism. In M. Bezem and J. F. Groote, editors, *TLCA'93*, LNCS 664, pp. 346–360. Springer-Verlag, 1993.
[35] B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.
[36] J. C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, pp. 408–423, 1974.
[37] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *OOPSLA*, pp. 43–55, 2001.
[38] W. Tait. Intensional interpretation of functionals of finite type i. *Journal of Symbolic Logic*, 32, 2:198–223, 1967.
[39] C. Wadsworth. The relation between computational and denotational properties for scott's D_∞-models of the lambda-calculus. *SIAM J. Comput.*, 5:488–521, 1976.
[40] T. Zhao, J. Palsberg, and J. Vitek. Lightweight Confinement for Featherweight Java. In *OOPSLA03*, pp. 135–148. ACM Press, 2003.