A three-valued type system for true positives detection in Java-like languages*

Davide Ancona
davide.ancona@unige.it

Federico Frassetto
federico.frassetto@dibris.unige.it

DIBRIS - Università di Genova, Italy

ABSTRACT

Soundness of type systems is an important property to guarantee the absence of certain kinds of runtime errors, that is, no false negatives can occur.

Unfortunately, for well-known theoretical limits, there are many programs that cannot be typed correctly, even though they will never manifest runtime errors, that is, false positives can occur.

Minimizing the rate of false positives makes static type analysis more effective, especially for dynamically typed languages. In this paper we propose a new approach to type systems, aiming to distinguish true from potentially false positives, and, thus, to provide useful hints on those lines of code that definitely contain a bug that sooner or later will occur.

To this aim, we define a three-valued type system for Featherweight Java which is sound in the usual sense, but can also distinguish true positives from potentially false ones.

Categories and Subject Descriptors

D.3.1 [**Programming languages**]: Formal Definitions and Theory

General Terms

Languages, Theory

Keywords

type systems, Java, true positives, three-valued logics

1. INTRODUCTION

Type systems are fundamental formal tools for static type analysis to guarantee early error detection in programs. To be useful, type systems must be sound to ensure that well-typed programs always enjoy certain important properties, which most of the times correspond to the absence of some kind of runtime errors. For instance, sound type systems for object-oriented languages prevent that errors like "field or method not found" can ever occur in well-typed programs. If a type system is sound, then *false negatives* will never occur; a false negative corresponds to a well-typed program whose execution can exhibit a problem that the type analysis was expected to prevent.

To be effective and implementable, a type system must provide an over-approximation of the semantics of the analyzed program. Loss of precision is experienced at the level of both control and data flow; in the former case, this happens because typechecking of an expression necessarily involves the typechecking of all its subexpressions, independently of whether or not they are reachable; in the latter case, this is directly connected with the notion of type, which is an abstraction of the notion of value, and with the notion of subtyping. Subtyping allows typechecking to be at the same time flexible and compositional, to avoid, for instance, typechecking the body of a method again for each distinct invocation; however, the cost for compositionality is a loss of precision in data flow analysis due to type widening.

Loss of precision of data flow analysis can be partially mitigated by introducing more precise types and parametric polymorphism (that is, generic classes and methods in object-oriented programming languages). However, for obvious reasons, loss of precision cannot be avoided; as a consequence, *false positives* are a necessary drawback of type analysis: some reported type errors will never manifest at runtime.

False positives are tolerated because their absence is only guaranteed by a complete type system, whose typechecking procedure is in general not decidable. However, a high average rate of false positives may compromise the effectiveness of a type system; smarter type systems can reduce this rate, and, indeed, this is the main solution adopted to tackle this problem.

In this paper we explore another possibility to reduce false positives, which can be usefully integrated with the most commonly followed solution of devising more expressive type systems. The idea we develop is based on the simple consideration that type systems do not exploit opportunities to classify positives (that is, detected type errors) into true and (potentially) false ones, even when this could be achieved with relatively modest efforts.

Let us consider, for instance, the expression `new C().m();`

*Partly funded by MIUR CINA - Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

FTfJP'17th Workshop on Formal Techniques for Java-like Programs, July 07 2015, Prague, Czech Republic.

ACM 978-1-4503-3656-7/15/07 ...$15.00
http://dx.doi.org/10.1145/2786536.2786539

if class `C` does not have method `m()`, then the typechecking of the expression simply fails, but an important detail is missing: such an error corresponds to a true positive. More precisely, in this case the type analysis could inform the user that when reached, the evaluation of the expression will always fail. Of course, the problem can never manifest if the expression can never be reached; anyway, in both cases a critical error has been detected: either the expression belongs to dead code, or it will eventually fail when evaluated.

If, instead, we consider the expression `x.m()`, where `x` has static type `C`, then the situation is different; if class `C` does not have method `m()`, then the typechecker should detect an error, which, however, should be classified as a potentially false positive, because `x` could denote an instance of a subclass of `C` which defines method `m()`.

Partitioning positives in true and potentially false ones has at least two advantages: it decreases the number of positives that could be false, and allows typecheckers to report errors at different levels of severity to focus the attention of developers on those parts of the program that surely contain errors, and, thus, require to be fixed with higher priority. This is especially useful when a static type analysis tool is used for detecting errors in programs written in dynamically typed scripting languages like JavaScript and Python. Indeed, differently from statically typed languages which are compiled only after typechecking has succeeded, type analysis in dynamic languages is optional and does not prevent the execution of programs for which errors have been reported. Under this point of view, error messages are simply considered as suggestions for potential bugs; however, if the analysis is able to identify lines of code for which an error will surely manifest sooner or later, then the developers can focus their attention on those fragments to try to solve the reported problems.

To allow a type system to distinguish between true and potentially false positives, a radical shift is needed: as has happened with shape analysis [7], the type system must be based on a three-valued logic. Given a type environment Γ, an expression e, and a type τ, the judgment e has type τ in Γ may have three different truth values: *true*, *maybe*, and *false*. Value *true* corresponds to the standard case, and the classical soundness result is expected to hold: either e diverges, or it evaluates to a value of type τ (true negative). Value *maybe* corresponds to the situation where e could have type τ (potentially false positive); in this case only a weak soundness result is expected to hold. Value *false* corresponds to the situation where e cannot have type τ; if this happens for all possible types τ, then a relative completeness result is expected to hold: whenever reached, e will fail (true positive).

As a concrete example, we define a three-valued type system for a slight variation of Featherweight Java (FJ) [6] which is sound in the usual sense, but can also distinguish between true positives from potentially false ones. In order to increase the number of detected true positives the type system uses also exact types [5]. The claims corresponding to the main properties of the type system (standard and weak soundness, and relative completeness) are expressed in terms of the approximating semantics [2] of FJ.

The paper is structured in the following way. Section 2 briefly introduces FJ and its big-step operational semantics; its three-valued type system is defined in Section 3. Section 4 contains the main claims concerning the type system, while Section 5 concludes and outlines directions for further investigation.

2. LANGUAGE SEMANTICS

In this section we present a slight variation of FJ and its big-step semantics.

The syntax is defined by the following extended BNF grammar.

$$\begin{array}{rcl} p & ::= & \overline{cd}^n\ e \\ cd & ::= & \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{fd}^n\ \overline{md}^k\ \} \\ fd & ::= & c\ f; \\ md & ::= & c_0\ m(\overline{c\ x}^n)\ \{e\} \\ e & ::= & \texttt{new}\ c(\overline{e}^n) \mid x \mid e.f \mid e_0.m(\overline{e}^n) \mid (c)\ e \end{array}$$

Assumptions: $n, k \geq 0$, inheritance is acyclic, names are distinct in class, method, field, and parameter declarations. Names of declared classes $\neq$ `Object`, names of declared parameters $\neq$ `this`.

The language coincides with FJ, except for the fact that constructors are declared implicitly.

Standard syntactic restrictions are implicitly imposed in the figure. Bars denote sequences of n items, where n is the superscript of the bar and the first index is 1. Sometimes this notation is abused, as in $\overline{f}^h = \overline{e'}^h$; which is a shorthand for $f_1 = e'_1; \ldots f_h = e'_h;$.

A program consists of a sequence of class declarations and a main expression. Types can only be class names.

A class declaration contains field and method declarations; in contrast with FJ, constructors are not declared, but every class is equipped with an implicit constructor with parameters corresponding to all fields, in the same order as they are inherited and declared.

Method declarations are standard; in the body, the target object can be accessed via the implicit parameter `this`, therefore all explicitly declared formal parameters must be different from `this`. Expressions include instance creation, variables, field selection, method invocation, and type casts.

The judgment $\Pi \vdash e \Rightarrow \mathbb{v}$ formalizes the big-step semantics, and is defined by the rules in Figure 1, and stating that in the evaluation environment Π, the expression e evaluates to the value $\mathbb{v}$.

The evaluation environment Π models the stack frame of the method that is currently executed, and it is a finite partial map from variables (corresponding to the implicit parameter `this` denoting the target object, and the explicitly declared formal parameters of the current method) to values. Object values are pairs $obj(c, [\overline{f}^n \mapsto \overline{\mathbb{v}}^n])$, where c is the class from which the object has been created, and $[\overline{f}^n \mapsto \overline{\mathbb{v}}^n]$ is a finite partial map associating the fields of the object with their corresponding values.

As usual, the judgment should be indexed over the collection of all class declarations contained in the program, however for brevity we leave implicit such an index in all judgments defined in the paper. The straightforward definitions of the auxiliary functions *fields* and *mcode* can be found in Figure 4.

3. TYPE SYSTEM

In this section we define a three-valued type system for FJ. Types used in the type systems are defined as follows.

$$\begin{array}{rcl} \tau & ::= & c^\nu \\ \nu & ::= & \epsilon \mid \circ \mid + \end{array}$$

Besides plain class names, which are the only allowed types

$$(\text{VAR})\ \frac{\Pi(x) = \mathbb{v}}{\Pi \vdash x \Rightarrow \mathbb{v}} \qquad (\text{NEW})\ \frac{\forall\, i \in \{1..n\}\ \Pi \vdash e_i \Rightarrow \mathbb{v}_i \quad \mathit{fields}(c) = \overline{c}^n\ \overline{f}^n}{\Pi \vdash \texttt{new}\ c(\overline{e}^n) \Rightarrow \mathit{obj}(c, [\overline{f}^n \mapsto \overline{\mathbb{v}}^n])} \qquad (\text{FLD})\ \frac{\Pi \vdash e \Rightarrow \mathit{obj}(c, [\overline{f}^n \mapsto \overline{\mathbb{v}}^n]) \quad 1 \le i \le n}{\Pi \vdash e.f_i \Rightarrow \mathbb{v}_i}$$

$$(\text{INV})\ \frac{\begin{array}{c}\forall\, i = 0..n. \Pi \vdash e_i \Rightarrow \mathbb{v}_i \quad \texttt{this} \mapsto \mathbb{v}_0, \overline{x}^n \mapsto \overline{\mathbb{v}}^n \vdash e \Rightarrow \mathbb{v} \\ \mathbb{v}_0 = \mathit{obj}(c_0, [\ldots]) \quad \mathit{mcode}(c_0, m) = \overline{x}^n.e\end{array}}{\Pi \vdash e_0.m(\overline{e}^n) \Rightarrow \mathbb{v}} \qquad (\text{CST})\ \frac{\Pi \vdash e \Rightarrow \mathbb{v} \quad \mathbb{v} = \mathit{obj}(c', [\ldots]) \quad c' \le c}{\Pi \vdash (c)\ e \Rightarrow \mathbb{v}}$$

Figure 1: Big-step operational semantics

$$(\mathit{refl}\le)\ \frac{}{\tau \le \tau} \qquad (\mathit{inh}\le)\ \frac{\texttt{class}\ c\ \texttt{extends}\ c''\ \{\ldots\} \quad c'' \le c'}{c \le c'} \qquad (\mathit{annot}\le)\ \frac{c_1 \le c_2 \quad \nu \ne \epsilon}{c_1^{\nu} \le c_2}$$

Figure 2: Subtyping rules

in the code, the type system employs two kinds of exact types, distinguished by an annotation of the corresponding class type; c° specifies values having exactly type c (that is, just the instances of c, without the instances of the subclasses of c), whereas c^+ is more precise than c° since it requires all fields of instances of c to contain a value having exactly the type declared for the field, and so on recursively. For instance, given the class declaration

```
class C extends Object { A f; }
```

and assuming that $D \le C$ and $B \le A$, and that A, B, and D do not declare fields, we have that `new` D(`new` A()) has type C, but neither C° nor C^+, `new` C(`new` B()) has type C, C°, but not C^+, and `new` C(`new` A()) has type C, C°, and C^+. The requirement for the type c^+ is very strong, but has been introduced to make the nominal type system more interesting, by allowing propagation of exact type information to the fields of an object, even though, in practice, this could be achieved more effectively on a per field basis with a structural type system.

The pretty standard subtyping rules are defined in Figure 2. Since plain class names are the only allowed types in the code, exact types cannot appear on the right hand side of the $\le$ relation symbol.

To support a three-valued logic, two main typing judgments are defined. The judgment $\Gamma \vdash e{:}\tau$ has the standard meaning, where Γ denotes a type environment, that is, a finite map from variables to types; the judgment is expected to be sound: in an evaluation environment compatible with Γ, the evaluation of e either diverges, or it returns a value of type τ. The judgment $\Gamma \vdash^? e{:}\tau$ means that in an evaluation environment compatible with Γ the evaluation of e may diverge, return a value or fail, but in case a value is provided, then that value will have type c. In other words, $\Gamma \vdash e{:}\tau$ and $\Gamma \vdash^? e{:}\tau$ correspond to the cases where typechecking returns the truth values *true* and *maybe*, respectively; for this reason, it can never happen[1] that both $\Gamma \vdash e{:}\tau$ and $\Gamma \vdash^? e{:}\tau$ are derivable. Finally, the truth value *false* corresponds to the fact that neither $\Gamma \vdash e{:}\tau$ nor $\Gamma \vdash^? e{:}\tau$ is derivable. Summarizing, $\Gamma \vdash e{:}\tau$ and $\Gamma \vdash^? e{:}\tau$ define a three-valued predicate *typecheck* as follows:

$$\mathit{typecheck}(\Gamma, e, \tau) = \begin{cases} \mathit{true} & \text{if } \Gamma \vdash e{:}\tau \\ \mathit{maybe} & \text{if } \Gamma \vdash^? e{:}\tau \\ \mathit{false} & \text{if } \Gamma \nvdash e{:}\tau \text{ and } \Gamma \nvdash^? e{:}\tau \end{cases}$$

If for all types τ neither $\Gamma \vdash e{:}\tau$ nor $\Gamma \vdash^? e{:}\tau$ is derivable, then the following relative completeness claim holds: when reached, the evaluation of e will always fail. The typing rules for $\Gamma \vdash e{:}\tau$ and $\Gamma \vdash^? e{:}\tau$ can be found in Figure 3.

Similarly to what happens for the operational semantics, all typing judgments are implicitly indexed over a class table containing all needed information on the classes declared in the program. Additionally, we implicitly refer to a method table *mt* consisting of the three functions *mtype*, *mtype*$^?$, and *mtype*$^\circ$ that we assume to be given, and whose meaning is explained below. Comments suggesting how a method table can be computed in practice for a given program can be found in Section 5.

In the rules the meta-variable κ can be instantiated only with either ?, or the empty string.

Rules (*pro*) and (*pro*?) define typechecking for a whole program; typechecking returns *true* if all classes of the program are well-typed, and its main expression e is well-typed in the empty type environment. It returns *maybe* if the typechecking of at least one class declaration, or the main expression returns *maybe*. The judgment $\vdash$ *mt ok* (defined in Figure 4) ensures in both rules that the method table *mt* (that is, the three functions *mtype*, *mtype*$^?$, and *mtype*$^\circ$) is consistent with the bodies of the methods declared in the program.

Similarly, rules (*cla*) and (*cla*?) define typechecking for class declarations; typechecking returns *true* if all methods of the checked class c are well-typed, and *fields*(c) is defined (the standard definition can be found in Figure 4), that is, no inherited field is redefined. It returns *maybe* if the typechecking of one method declaration returns *maybe*, providing that *fields*(c) is defined; this last condition ensures that the objects of the class will have a valid structure, hence it is essential also for the *maybe* case.

Typechecking of methods (rules (*met*) and (*met*?)) rely on the functions *mtype* and *mtype*$^?$ defined by the implicit method table *mt*; $\mathit{mtype}(c, m, \overline{c}^n \to c_0)$ is a more compact notation for $\mathit{mtype}(c, m) = \overline{c}^n \to c_0$, to mean that class c has a method m of arity n that when invoked with arguments compatible with the parameters of type $\overline{c}^n$, correctly returns a value of type c_0. Similarly, $\mathit{mtype}^?(c, m, \overline{c}^n \to c_0)$ means

[1]The proof of this claim, omitted for space limitation, can proceed by induction on the typing rules, and exploits the hypothesis that if $\mathit{mtype}(c_0, m, \overline{c}^n \to c)$ holds, then $\mathit{mtype}^?(c_0, m, \overline{c}^n \to c')$ does not hold for all c'.

$$(pro)\ \frac{\forall i \in \{1..n\}\ \vdash cd_i\ ok \quad \vdash mt\ ok\ \emptyset \vdash e{:}\tau}{\vdash \overline{cd}^n\ e\ ok} \qquad (pro?)\ \frac{\forall i \in \{1..n\}\ \vdash^{\kappa_i} cd_i\ ok \quad \vdash mt\ ok\ \emptyset \vdash^{\kappa} e{:}\tau\ \exists i \in \{1..n\}\ \kappa_i =? \vee \kappa =?}{\vdash^{?} \overline{cd}^n\ e\ ok}$$

$$(cla)\ \frac{\forall i \in \{1..k\}\ c \vdash md_i\ ok \quad \mathit{fields}(c)\ \text{defined}}{\vdash \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \overline{fd}^n\ \overline{md}^k\ \}\ ok} \qquad (cla?)\ \frac{\forall i \in \{1..k\}\ c \vdash^{\kappa_i} md_i\ ok \quad \mathit{fields}(c)\ \text{defined} \quad \exists i \in \{1..n\}\ \kappa_i =?}{\vdash^{?} \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \overline{fd}^n\ \overline{md}^k\ \}\ ok}$$

$$(met)\ \frac{\mathit{mtype}(c, m, \overline{c}^n \to c_0) \quad \mathit{override}(c, m, \overline{c}^n, c_0)}{c \vdash c_0\ m(\overline{c}^n\ \overline{x}^n)\ \{e\}\ ok} \qquad (met?)\ \frac{\mathit{mtype}^?(c, m, \overline{c}^n \to c_0) \quad \mathit{override}(c, m^?, \overline{c}^n, c_0)}{c \vdash^? c_0\ m(\overline{c}^n\ \overline{x}^n)\ \{e\}\ ok}$$

$$(var)\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x{:}\tau} \qquad (new^+)\ \frac{\forall i \in \{1..n\}\ \Gamma \vdash e_i{:}c_i^+ \quad \mathit{fields}(c) = \overline{c}^n\ \overline{f}^n}{\Gamma \vdash \texttt{new}\ c(\overline{e}^n){:}c^+}$$

$$(new)\ \frac{\forall i \in \{1..n\}\ \Gamma \vdash e_i{:}c_i^{\nu_i} \quad \mathit{fields}(c) = \overline{c'}^n\ \overline{f}^n \quad \forall i \in \{1..n\}\ c_i \le c_i' \quad \exists i \in \{1..n\}\ \nu_i \neq + \vee c_i \neq c_i'}{\Gamma \vdash \texttt{new}\ c(\overline{e}^n){:}c^\circ}$$

$$(new?)\ \frac{\forall i \in \{1..n\}\ \Gamma \vdash^{\kappa_i} e_i{:}\tau_i \quad \mathit{fields}(c) = \overline{c}^n\ \overline{f}^n \quad \exists i \in \{1..n\}\ \kappa_i =? \vee \tau_i \not\le c_i}{\Gamma \vdash^? \texttt{new}\ c(\overline{e}^n){:}c^\circ}$$

$$(fld)\ \frac{\Gamma \vdash e{:}c^\nu \quad \mathit{fields}(c) = \overline{c}^n\ \overline{f}^n \quad 1 \le i \le n \quad \nu = + \Rightarrow \nu' = + \quad \nu \neq + \Rightarrow \nu' = \epsilon}{\Gamma \vdash e.f_i{:}c_i^{\nu'}}$$

$$(fld?)\ \frac{\Gamma \vdash^{\kappa} e{:}c^\nu \quad \mathit{fields}(c) = \overline{c}^n\ \overline{f}^n \quad \kappa =? \vee (\nu = \epsilon \wedge f \notin \{f_1, \ldots, f_n\})}{\Gamma \vdash^? e.f{:}\texttt{Object}}$$

$$(cast\text{-}wide)\ \frac{\Gamma \vdash^{\kappa} e{:}c_1^{\nu_1} \quad c_1 \le c_0 \quad c_0 = c_1 \Rightarrow \nu_0 = \nu_1 \quad c_0 \neq c_1 \Rightarrow \nu_0 = \epsilon}{\Gamma \vdash^{\kappa} (c_0)\ e{:}c_0^{\nu_0}} \qquad (cast\text{-}narrow)\ \frac{\Gamma \vdash^{\kappa} e{:}c_1 \quad c_0 \le c_1 \quad c_0 \neq c_1}{\Gamma \vdash^? (c_0)\ e{:}c_0}$$

$$(inv)\ \frac{\forall i \in \{0..n\}\ \Gamma \vdash e_i{:}\tau_i \quad \vdash \tau_0.m(\overline{\tau}^n){:}\tau}{\Gamma \vdash e_0.m(\overline{e}^n){:}\tau} \qquad (inv?)\ \frac{\forall i \in \{0..n\}\ \Gamma \vdash^{\kappa_i} e_i{:}\tau_i \quad \vdash^{\kappa} \tau_0.m(\overline{\tau}^n){:}\tau \quad \exists i \in \{0..n\}\ \kappa_i =? \vee \kappa =?}{\Gamma \vdash^? e_0.m(\overline{e}^n){:}\tau}$$

Figure 3: Nominal type system

that class c has a method m of arity n that even when invoked with arguments compatible with the parameters of type $\overline{c}^n$, might fail to complete or might return a value non compatible with c_0.

For a method declaration to be well-typed, the usual constraints on method overriding have to be verified (with the auxiliary predicate *override* defined in Figure 4); if a method with the same name is declared in a subclass, then its type must be a subtype of the overridden method: the arity must be the same (overloading is not supported), parameter types can be widened, while the return type can be narrowed. In case the typechecking of the method declaration returns *maybe* (case $\mathit{override}(c, m^?, \overline{c}^n, c_0)$), the requirements for correct overriding are stricter: the typechecking of the overridden method, if present, must return *maybe* as well. To see why, let us consider the following example:

```
class C extends Object {
    D m1(C x){ return x }
    C m2(C x){ return x }
}
class D extends C {
    D m1(C x){ return new D() }
    D m2(C x){ return x }
}
```

Since $\texttt{C} \not\le \texttt{D}$, the typechecking of method `m1` in class `C`, and `m2` in class `D` returns *maybe*, whereas the typechecking of the other two methods returns *true*. The overriding rules concerning method `m1` are verified; indeed, the typechecking of `x.m1(x)` returns *maybe* if `x` has static type `C`, since method `m1` of class `C` might be invoked, while it returns *true* if `x` has static type `D`, since method `m1` of class `C` may not be invoked. The overriding rules concerning method `m2` are not verified since the typechecking of the overriding method returns *maybe*, whereas the typechecking of the overridden method returns *true*; indeed, if `x` has static type `C`, then the typechecking of `x.m2(x)` would return *true*, but method `m2` of class `D` might be invoked, and, hence, typechecking should return *maybe*.

The typechecking of variables (rule (*var*)) may only return either *true* or *false*, but not *maybe*, because the evaluation of x will always fail if x is not in scope.

For instance creation there are three rules: (new^+), (*new*), and (*new*?); for the first two rules, typechecking returns *true*, but the derived type for (new^+) is more precise. Indeed, if all the argument expressions $\overline{e}^n$ have the exact type c_i^+, where c_i coincides with the type of the corresponding field, then the exact type c^+ can be deduced for the whole expression.

If at least one argument expression does not have an exact type annotated with $+$, and coinciding with the type of the corresponding field, then the less precise type c° can be deduced, providing that the types of the argument expressions are subtypes of the declared types for the corresponding fields. For instance, let us consider the following class declarations and assume that class `C` does not have fields:

```
class A extends Object { B f; }
class B extends Object { C f; }
```

The type $\texttt{A}^+$ can be deduced for `new A(new B(new C()))`, whereas, if we assume that `D` is a subclass of `C` that does not declare fields, then the type $\texttt{A}^\circ$, but not $\texttt{A}^+$, can be deduced for `new A(new B(new D()))` (and `new B(new D())` and `new D()` have type $\texttt{B}^\circ$ and $\texttt{D}^+$, respectively).

Finally, according to rule (*new*?), the typechecking of the

$$\frac{}{\mathit{fields}(\texttt{Object}) = \epsilon} \qquad \frac{\begin{array}{c}\mathit{fields}(c') = \overline{\tau}^m\ \overline{f}^m \\ \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \overline{c}^n\ \overline{g}^n; \overline{md}^k\} \\ \{f_1, \ldots, f_m\} \cap \{g_1, \ldots g_n\} = \emptyset\end{array}}{\mathit{fields}(c) = \overline{\tau}^m \overline{f}^m, \overline{c}^n\ \overline{g}^n} \qquad \frac{\begin{array}{c}\mathit{mtype}(c_0, m, \overline{c}^n \to c) \Rightarrow \neg \mathit{mtype}^?(c_0, m, \overline{c}^n \to c') \\ \mathit{mtype}(c_0, m, \overline{c}^n \to c) \Rightarrow \vdash \mathit{mtype}(c_0, m, \overline{c}^n \to c)\ ok \\ \mathit{mtype}^?(c_0, m, \overline{c}^n \to c) \Rightarrow \vdash \mathit{mtype}^?(c_0, m, \overline{c}^n \to c)\ ok \\ \mathit{mtype}^\circ(c, m, \mu\tau) \Rightarrow \vdash \mathit{mtype}^\circ(c, m, \mu\tau)\ ok\end{array}}{\vdash mt\ ok}$$

$$\frac{\texttt{class}\ c_0\ \texttt{extends}\ c_0'\ \{\ \overline{fd}^n\ \overline{md}^k\ c\ m(\overline{c\ x}^n)\ \{e\}\ \overline{md'}^{k'}\ \}}{\mathit{mdec}(c_0, m) = c_0.\overline{c}^n\ \overline{x}^n.e{:}c} \qquad \frac{\begin{array}{c}\mathit{mdec}(c_0', m) = c_0''.\overline{c}^n\ \overline{x}^n.e{:}c \\ \texttt{class}\ c_0\ \texttt{extends}\ c_0'\ \{\ \overline{fd}^n\ \overline{md}^k\} \\ m\ \text{not declared in}\ \overline{md}^k\end{array}}{\mathit{mdec}(c_0, m) = c_0''.\overline{c}^n\ \overline{x}^n.e{:}c}$$

$$\frac{\mathit{mdec}(c_0, m) = c_0'.\overline{c}^n\ \overline{x}^n.e{:}c}{\mathit{mcode}(c_0, m) = \overline{x}^n.e} \qquad \frac{\mathit{mdec}(c_0, m) = c_0'.\overline{c}^n\ \overline{x}^n.e{:}c \quad \texttt{this}{:}c_0', \overline{x}^n{:}\overline{c}^n \vdash e{:}\tau \quad \tau \leq c}{\vdash \mathit{mtype}(c_0, m, \overline{c}^n \to c)\ ok}$$

$$\frac{\mathit{mdec}(c_0, m) = c_0'.\overline{c}^n\ \overline{x}^n.e{:}c \quad \texttt{this}{:}c_0', \overline{x}^n{:}\overline{c}^n \vdash^\kappa e{:}\tau \quad \kappa = ? \vee \tau \not\leq c}{\vdash \mathit{mtype}^?(c_0, m, \overline{c}^n \to c)\ ok}$$

$$\frac{\begin{array}{c}\mathit{mdec}(c, m) = c'.\overline{\tau_0}^n\ \overline{x}^n.e{:}\tau \quad \mathit{mtype}(c, m, \overline{\tau_0}^n \to \tau_0) \\ \forall i \in \{1..k\}\ \texttt{this}{:}c', \overline{x}^n{:}\overline{\tau_i}^n \vdash e{:}\tau_i \quad \forall i, j \in \{0..k\}\ i \neq j \Rightarrow \overline{\tau_i}^n \neq \overline{\tau_j}^n\end{array}}{\vdash \mathit{mtype}^\circ(c, m, \bigwedge_{i \in \{0..k\}} \overline{\tau_i}^n \to \tau_i)\ ok}$$

$$\frac{\texttt{class}\ c_0\ \texttt{extends}\ c_0'\ \{\ \ldots\ \} \quad \mathit{mcode}(c_0', m)\ \text{undefined}}{\mathit{override}(c_0, m^\kappa, \overline{c}^n, c)} \qquad \frac{\texttt{class}\ c_0\ \texttt{extends}\ c_0'\ \{\ \ldots\ \} \quad \mathit{mcode}(c_0', m) = \overline{c'}^n\ \overline{x}^n.e{:}c' \quad \overline{c'}^n \leq \overline{c}^n\ \text{and}\ c \leq c'}{\mathit{override}(c_0, m, \overline{c}^n, c))}$$

$$\frac{\texttt{class}\ c_0\ \texttt{extends}\ c_0'\ \{\ \ldots\ \} \quad \mathit{mcode}(c_0', m) = \overline{c'}^n\ \overline{x}^n.e{:}c' \quad c_0' \vdash^?\ c'\ m(\overline{c'}^n\ \overline{x}^n)\ \{e\}\ ok \quad \overline{c'}^n \leq \overline{c}^n\ \text{and}\ c \leq c'}{\mathit{override}(c_0, m^?, \overline{c}^n, c))}$$

$$(\mathit{meth\text{-}inv}^\circ)\ \frac{\nu \neq \epsilon \quad \mathit{mtype}^\circ(c, m, \mu\tau) \quad \mathit{best\_matches}(\overline{\tau}^n, \mu\tau) = \{\tau\}}{\vdash c^\nu.m(\overline{\tau}^n){:}\tau}$$

$$(\mathit{meth\text{-}inv})\ \frac{\mathit{mtype}(c_0, m, \overline{c}^n \to c) \quad \overline{\tau}^n \leq \overline{c}^n}{\vdash c_0.m(\overline{\tau}^n){:}c} \qquad (\mathit{meth\text{-}inv?})\ \frac{\begin{array}{c}(\mathit{mtype}^\kappa(c_0, m, \overline{c}^n \to c) \wedge (\kappa = ? \vee \exists i \in \{1..n\} \tau_i \not\leq c_i)) \\ \vee \mathit{mdec}(c_0, m)\ \text{undefined}\end{array}}{\vdash^?\ c_0.m(\overline{\tau}^n){:}\texttt{Object}}$$

$$\mathit{best\_matches}(\overline{\tau}^n, \textstyle\bigwedge_{i \in \{1..n\}} \overline{\tau_i}^n \to \tau_i) = \{\tau_i \mid i \in \{1..n\}, \overline{\tau}^n \leq \overline{\tau_i}^n, \forall j \in \{1..n\} (j \neq i, \overline{\tau}^n \leq \overline{\tau_j}^n) \Rightarrow \overline{\tau_j}^n \not\leq \overline{\tau_i}^n\}$$

Figure 4: Auxiliary functions and judgments

whole expression returns *maybe*, if the typechecking of one argument expressions returns *maybe*, or if the invariant that argument types must be subtype of the corresponding field types is broken. If $\mathit{fields}(C)$ is undefined or returns a number of fields different from the number of argument expressions, the typechecking returns *false* (hence, fails), because in such situations the evaluation of $\texttt{new}\ c(\overline{e}^n)$ always fails.

For field selection $e.f$ (rules (*fld*) and (*fld?*)), the typechecking returns *true* if the typechecking of e returns *true*, and for the corresponding class type c, *fields* is defined and returns a list of fields including f. The class type of the whole expression is the declared type for f, while the annotation depends on the annotation of the type of e: the annotation is preserved only in case of $+$, in all other cases a non exact type is derived.

The typechecking of the whole expression returns *maybe* if either the typechecking of e returns *maybe*, or $\mathit{fields}(c)$ is defined, but does not include the field f and the class type of e does not carry any annotation; if the typechecking of e returns *maybe*, then some subtype constraint that is expected to be verified for instance creation or method invocation might be broken, therefore the derived type for the whole expression can only be the top type `Object`; analogously, if f cannot be found in $\mathit{fields}(c)$, and the type of e is c (without any annotation), then e could denote an instance of a subclass of c that could have field f, but no type information can be deduced for its type, except for the obvious top type `Object`. Finally, if $\mathit{fields}(c)$ is undefined, or f is not included in $\mathit{fields}(c)$, and e has type c^ν, with $\nu \neq \epsilon$, then the typechecking of the whole expression returns *false* because the evaluation of $e.f$ will always fail.

For casting (rules (*cast-wide*) and (*cast-narrow*)), there are two different rules, the former dealing with the situation where the type of e will be always compatible with the type c_0 required by the cast, and the latter with the case where the type of e could be compatible with the type c_0. For the first rule, the validity of $c_1 \leq c_0$ ensures that the cast is safe. However, the truth value returned by the typechecking of the whole expression always coincides with that returned by the typechecking of e. The annotation of the class type for e is propagated to the class type for the whole expression if $c_0 = c_1$ (identity conversion), otherwise the widening conversion implies a loss of type information, and the derived type for the whole expression cannot carry any annotation.

Rule (rules (*cast-narrow*) deals with the disjoint case $c_0 \leq c_1$, $c_0 \neq c_1$, corresponding to type narrowing. In this case the returned truth value will always be *maybe*, since there is no guarantee that the dynamic typecheck will succeed; furthermore, e could denote an instance of a subclass of c_0, therefore no type annotation can be derived for the class type of the whole expression. Finally, if either C_0 and C_1 are not comparable ($c_1 \not\leq c_0$ and $c_0 \not\leq c_1$), or the type of e is exact (with annotation $+$ or $\circ$) and $c_1 \not\leq c_0$, then the typechecking of the whole expression returns *false* because

its evaluation will always fail.

The typing rules for method invocation (rules (*inv*) and (*inv*?)) use an auxiliary three-valued judgment, defined in Figure 4; $\vdash \tau_0.m(\overline{\tau}^n):\tau$ means that an invocation of method m on a target object of type τ_0 with n arguments of type $\overline{\tau}^n$ is always type safe (truth value *true*) and always returns a value of type τ; similarly, $\vdash^? \tau_0.m(\overline{\tau}^n):\tau$ means that an invocation of method m on a target object of type τ_0 with n arguments of type $\overline{\tau}^n$ might be type safe (truth value *maybe*) and, if so, the returned value is guaranteed to have type τ. Rule (*inv*) is applicable when typechecking returns *true*, whereas if the typechecking of some subexpression returns *maybe* or the judgment $\vdash^? \tau_0.m(\overline{\tau}^n):\tau$ is derivable, then the typechecking of the whole expression returns *maybe* (rule (*inv*?)).

The judgments $\vdash \tau_0.m(\overline{\tau}^n):\tau$ and $\vdash^? \tau_0.m(\overline{\tau}^n):\tau$ are defined by the three rules (*meth-inv*$^\circ$), (*meth-inv*), and (*meth-inv*?). The first rule deals with the situation where the most precise type information can be deduced for the target object; indeed, if the type of the target is exact (with annotation $+$ or $\circ$), then the exact method that will be invoked is known statically; in this case, a more precise type for the method, not subjected to the rules for overriding, can be used, to make typechecking more accurate. Such a type is returned by the function *mtype*$^\circ$ defined by the method table mt, and whose consistency is checked by the judgment $\vdash mtype^\circ(c, m, \bigwedge_{i\in\{0..k\}} \overline{\tau_i}^n \to \tau_i)\ ok$ defined in Figure 4; the function is expected to return a conjunction $\bigwedge_{i\in\{0..k\}} \overline{\tau_i}^n \to \tau_i$ of method types (that is, arrow types), all having the same number n of parameters, where the first method type[2] must always coincide with the default type of the method, as declared in the method definition $(mdec(c, m) = c'.\overline{\tau_0}^n\ \overline{x}^n.e:\tau)$, and all other types must be consistent w.r.t. the method body. Finally, the argument types in the conjunction must be pairwise disjoint to avoid ambiguities.

An invocation of method m of type $\bigwedge_{i\in\{0..k\}} \overline{\tau_i}^n \to \tau_i$ for arguments of type $\overline{\tau_i}^m$ is correct only if $n = m$, and there exists a unique member $\overline{\tau_i}^n \to \tau_i$ which is compatible with the type arguments $(\overline{\tau_i}^m \leq \overline{\tau_i}^n \to \tau_i)$ and is minimal (there are no more specific applicable types, as defined by the auxiliary function *best_matches* in Figure 4).

Rule (*meth-inv*) corresponds to the standard rule for typechecking method invocation; the type c_0 of the target is not exact, hence the exact method that will be executed cannot be known statically, since an overriding method defined in a subclass of c_0 could be called. The function *mtype* ensures that the types declared in the method definition for the parameters and the returned value are correct, that is, the typechecking of the body returns *true* in the type environment defined by the parameter declaration for a subtype of the return type. The typechecking succeeds if the types of the arguments are compatible with the types of the corresponding parameters.

The typechecking of method invocation returns *maybe* (rule (*meth-inv*?)), if either $mtype^?(c_0, m, \overline{c}^n \to c)$ holds, or $mtype(c_0, m, \overline{c}^n \to c)$ holds and there exists an argument whose type is not compatible with the type of the corresponding parameter, or the class c_0 of the target object does not have any method m ($mdec(c_0, m)$ undefined). Differently from casts, passing an argument that will always be incompatible with the formal parameter of a method does not imply that its invocation will fail. Finally, we recall that FJ does not allow method overloading, therefore if c_0 has a method m with the wrong number of parameters, then all subclasses of c_0 will have method m with the wrong number of parameters.

Because some subtype constraint is broken, or no method declaration can be found, in this case the unique type that can be correctly derived for the whole expression is `Object`.

If the type of the target object is exact, but no well-typed method m (that is, the typechecking of the body returns *true*) with n arguments can be found, or the type of the class is not exact and a method m with the wrong number of parameters is found, then the typechecking of the method invocation returns *false* because its evaluation will always fail.

[2]For sake of simplicity, we follow the convention that the default method type is always the first one, even though this is not strictly necessary, since type conjunction is commutative.

4. FORMAL RESULTS

In this section we formalize the main properties of the type system, and provide some proof sketches.

4.1 Standard soundness

When typechecking returns the truth value *true* (judgment $\Gamma \vdash e:\tau$) the standard soundness property is expected to hold. As shown in previous work [1, 2], the soundness claim can be expressed and proved by introducing the notion of approximating semantics.

The rules in Figure 1 are extended with the rule (APPROX) allowing arbitrary approximation of the proof trees. In this way, infinite proof trees corresponding to diverging computations can be approximated by an infinite sequence of finite proof trees, solving the typical issue concerning the inability of big-step operational semantics to capture non terminating computations; with approximating semantics the proof of soundness can be done by simple arithmetic induction.

$$\text{(APPROX)}\ \frac{}{\Pi \vdash e \Rightarrow \mathbb{v}}$$

The auxiliary judgment $\Pi \vdash^{\approx}_n e \Rightarrow \mathbb{v}$ is introduced to specify that the judgment $\Pi \vdash e \Rightarrow \mathbb{v}$ can be (inductively) derived from the system of semantic rules extended with rule (APPROX), by possibly using the new rule, but only at depth $d \geq n$ in the proof tree. Intuitively, the subscript n in $\Pi \vdash^{\approx}_n e \Rightarrow \mathbb{v}$ specifies the precision of the approximation: the higher n, the more accurate is the approximation. If $\Pi \vdash^{\approx}_i e \Rightarrow \mathbb{v}_i$ for all $i \in \mathbb{N}$, then the evaluation of e in Π either converges, hence, there exists a proof tree of depth n for $\Pi \vdash e \Rightarrow \mathbb{v}$ and $\mathbb{v}_i = \mathbb{v}$ for all $i > n$, or diverges, but cannot go wrong.

For formulating the claim of soundness the usual judgment $\mathbb{v} \in \tau$ for typing values (whose rules have been omitted for space limitations) and the agreement relation between evaluation and type environments are needed: $\Pi \in \Gamma$ iff $dom(\Gamma) \subseteq dom(\Pi)$ and for all $x \in dom(\Gamma)$ $\Pi(x) \in \Gamma(x)$.

THEOREM 4.1 (STANDARD SOUNDNESS). *If for all* $i \in \{1..n\}$ $\vdash cd_i\ ok$, $\vdash mt\ ok$, $\Gamma \vdash e:\tau$, *and* $\Pi \in \Gamma$ *in* $\overline{cd}^n$, *then for all* $k \in \mathbb{N}$ *there exists* $\mathbb{v}_k$ *s.t.* $\Pi \vdash^{\approx}_k e \Rightarrow \mathbb{v}_k$ *in* $\overline{cd}^n$, *and* $\mathbb{v}_k \in \tau$.

4.2 Weak soundness

In case typechecking returns the *maybe* truth value, a much weaker property can be proved, stating that if a value is returned, then it must agree with the derived type for the expression.

The claim can be easily formulated in terms of the approximating semantics, as done for the standard soundness result. In this case typechecking is allowed to return *maybe* not only for the main expression, but also for the classes declared by the program, and the agreement relation between evaluation and type environments: $\Pi \in^{?} \Gamma$ iff $dom(\Gamma) \subseteq dom(\Pi)$ and there exists $x \in dom(\Gamma)$ s.t. $\Pi(x) \notin \Gamma(x)$.

THEOREM 4.2 (WEAK SOUNDNESS). *If for all* $i \in \{1..n\}$ $\vdash^{\kappa_i} cd_i\ ok$, $\vdash mt\ ok$, $\Gamma \vdash^{\kappa} e{:}\tau$, $\Pi \in^{\kappa'} \Gamma$ *in* $\overline{cd}^n$, *and for all* $k \in \mathbb{N}$ *there exists* $\mathbb{v}_k$ *s.t.* $\Pi \vdash^{\approx}_k e \Rightarrow \mathbb{v}_k$ *in* $\overline{cd}^n$, *then* $\mathbb{v}_k \in \tau$.

4.3 Relative completeness

The three-valued type system allows formulation of a completeness result that holds if one assumes that the ill-typed expression is reachable.

THEOREM 4.3 (RELATIVE COMPLETENESS). *If for all* $i \in \{1..n\}$ $\vdash^{\kappa_i} cd_i\ ok$, $\vdash mt\ ok$, $\Pi \in^{\kappa'} \Gamma$, *and for all types* τ $\Gamma \not\vdash e{:}\tau$ *and* $\Gamma \not\vdash^{?} e{:}\tau$, *then there exists* $k_0 \in \mathbb{N}$ *s.t. for all* $k > k_0$ *there is no* $\mathbb{v}$ *s.t.* $\Pi \vdash^{\approx}_k e \Rightarrow \mathbb{v}$ *in* $\overline{cd}^n$.

Let us consider, for instance, the following class declaration.

```
class C extends Object {
    C m1(){return this.m()}
    C m2(C x,C y){return x}
}
```

If e_{fail} denotes an expression for which typechecking fails, then the typechecking of e`=new C().m2(new C().m1(),`e_{fail}`)` returns *false*, that is, $\emptyset \not\vdash e{:}\tau$ and $\emptyset \not\vdash^{?} e{:}\tau$ for all types τ. Failure of e is generated from the subexpression e_{fail}, however, since `new C().m1()` diverges and arguments are evaluated from left to right, the evaluation of e diverges and, hence, does not fail. Nevertheless, the type system detects that the evaluation goes wrong under the assumption that e_{fail} is reachable. Interestingly enough, for languages like FJ, where there are no conditional expressions, the approximating semantics always fails if there exists a subexpression that would fail when reached; for this specific example, an infinite sequence of approximating proof trees can be built for e, only if the same holds for `new C()` and e_{fail}, but this property does not hold for e_{fail}, therefore there exists k_0 s.t. for all $k > k_0$, there is no value $\mathbb{v}$ s.t. $\emptyset \vdash^{\approx}_k e \Rightarrow \mathbb{v}$.

However, for languages with conditional expressions, or, more in general, short-circuiting operators, the statement holds only if it is existentially quantified over the subexpressions of e (e included). For instance, the typechecking of e'`=if(true) then 1 else` e_{fail} fails, but for all $k \in \mathbb{N}$, $\emptyset \vdash^{\approx}_k e' \Rightarrow 1$. However, there exists a subexpression e'' of e' (e_{fail} in this case) s.t. there exists k_0 s.t. for all $k > k_0$, there is no value $\mathbb{v}$ s.t. $\emptyset \vdash^{\approx}_k e'' \Rightarrow \mathbb{v}$.

5. CONCLUSION AND FUTURE WORK

We have defined a type system for FJ based on a three-valued logic which is able to distinguish true positives from potentially false ones, and, therefore, is able to produce more informative and useful type errors.

Three main claims about the type system have been formulated in terms of the approximating big-step semantics of the language: standard soundness (when typechecking returns the truth value *true*), weak soundness (when *maybe* is returned), and relative completeness (when *false* is returned).

To our knowledge, this is the first attempt to define a type system based on a three-valued logic to detect true positives. For simplicity, we have integrated our idea with the nominal type system of FJ, even though we expect that more interesting results could be achieved with structural types.

One of the main issues that have to be addressed to make this approach effective is how to get the right balance between compositionality and precision, that is, how a method table can be computed in practice for a given program: on one hand one need to propagate exact type information to the body of the method in order to maximize the detection of true positives (and to get more accurate typing, of course), on the other one, compositional typechecking of methods would be preferable.

To leave open all possibilities, for the proposed type system we have deliberately chosen a non algorithmic definition based on the notion of method table, and of its three components *mtype*, $mtype^{?}$, and $mtype^{\circ}$. Whereas the functions *mtype*, and $mtype^{?}$ can be easily computed thanks to the type information in method headers, computing $mtype^{\circ}$ is more challenging. A simple algorithmic solution could consist in typechecking method bodies once for all for a fixed number of combinations of type parameters, automatically deduced[3] from the declared types. More sophisticated solutions can rely on type constraint generation and parametric polymorphism [4, 3].

6. REFERENCES

[1] D. Ancona. Soundness of Object-Oriented Languages with Coinductive Big-Step Semantics. In J. Noble, editor, *ECOOP 2012 - Object-Oriented Programming*, volume 7313, pages 459–483. Springer, 2012.

[2] D. Ancona. How to prove type soundness of Java-like languages without forgoing big-step semantics. In *FTfJP'14*, pages 1:1–1:6. ACM, 2014.

[3] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *POPL 2005*, pages 26–37, 2005.

[4] D. Ancona and E. Zucca. Principal typings for Java-like languages. In *POPL 2004*, pages 306–317, 2004.

[5] K. B. Bruce and J. N. Foster. LOOJ: weaving LOOM into Java. In *ECOOP 2004*, pages 389–413, 2004.

[6] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

[7] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, 2013.

[3] For instance, for a method with two parameters with declared types c_1 and c_2, the body could be typechecked also when both parameters have types c_1^{+}, and c_2^{+} and c_1° and c_2°.