Semantic Foundations and Inference of Non-null Annotations

Laurent Hubert[1], Thomas Jensen[1], and David Pichardie[2]

[1] CNRS/IRISA, France
[2] INRIA Rennes - Bretagne Atlantique/IRISA, France

Abstract. This paper proposes a semantics-based automatic null pointer analysis for inferring non-null annotations of fields in object-oriented programs. The analysis is formulated for a minimalistic OO language and is expressed as a constraint-based abstract interpretation of the program which for each field of a class infers whether the field is definitely non-null or possibly null after object initialization. The analysis is proved correct with respect to an operational semantics of the minimalistic OO language. This correctness proof has been machine checked using the Coq proof assistant. We also prove the analysis complete with respect to the non-null type system proposed by Fähndrich and Leino, in the sense that for every typable program the analysis is able to prove the absence of null dereferences without any hand-written annotations. Experiments with a prototype implementation of the analysis show that the inference is feasible for large programs.

1 Introduction

A common source of exceptional program behaviour is the dereferencing of null references (also called null pointers), resulting in segmentation faults in C or null pointer exceptions in Java. Even if such exceptions are caught, the presence of exception handlers creates an additional amount of potential branching which in turn implies that: 1) fewer optimizations are possible and 2) verification is more difficult (bigger certification conditions, implicit flow in information flow verification, etc.). Furthermore, the Java virtual machine is obliged to perform run-time checks for non-nullness of references when executing a number of its bytecode instructions, thereby incurring a performance penalty.[3] For all these reasons, a static program analysis which can guarantee before execution of the program that certain references will definitely be non-null is useful.

Some non-null type systems for the Java language have been proposed [6, 15]. Although non-null annotations are not used by the compiler yet, this could likely change in the future, in which case it becomes crucial to prove the soundness of the logic underlying the type checker. Furthermore, although non-null annotations are not yet mandatory, automatic type inference would help to retrofit

[3] Although hardware traps are used for free whenever possible, explicit non-nullness tests are still required as explained in [13]

legacy code, to lower the annotation burden on the programmer, to document the code and to verify existing code.

While some object oriented languages ensure all fields are initialized and objects are not read before being fully initialized, this is not the case in Java. More precisely, there are three aspects that complicate non-nullness analysis: 1) fields can be accessed during object construction (before or after their initialization) which means all fields contain null value before their first assignment; 2) no guarantee is given on the initialization of fields at the end of the constructor so if a field has not been initialized it contains a null value; and 3) an object being initialized can be stored in fields or passed as an argument of methods.

The first aspect means a naive flow-insensitive heap abstraction is not sufficient as all fields are null at some stage and hence would be annotated as possibly null. The second aspect makes a special initialization analysis necessary in order to track fields that are certain to be initialized during the construction. Those fields can safely be considered as non-null unless they are explicitly assigned a null value. All other fields might not have been initialized and must be considered as possibly null.

The third aspect was observed in [6] and concerns the problem where a virtual method `A.m()` is being redefined in a sub-class `B` by a method referencing a field `f` that is local to the class `B`. If the constructor of a super-class `A` executes `this.m()` on an object of class `B`, it calls a method that dereferences field `f` before the constructor of `B` has had the possibility of initializing this field. To solve this problem, references to objects under construction need to be tracked, e.g. by a tag indicating their state.

Related Work. Freund and Mitchell have proposed in [9] a *type system* combined with a data-flow analysis to ensure the correct initialization of objects. The goal is to formalize the initialization part of Java bytecode verification. In this respect it is different from our analysis, which is focused on field initialization and on their nullness property. Fähndrich and Leino in [6] have proposed another *type system* also combined with a data-flow analysis to ensure a correct manipulation of references with respect to a nullness property. This system is presented in Sect. 6 where we compare our inference analysis to their type system and give examples of how our analysis infers more precise types than what their system is able to check. More recently, Fähndrich and Xia have proposed another *type system* introducing *delayed* initialization [7]. It generalizes the previous one and allows to prove some properties that our analysis cannot, like initialization of circular data structures, but it has also the same loss of precision discussed in Sect. 6.

Some works are focused on *local type inference*, *i.e.* inferring nullness property for blocks of code from the guards. This is notably the case of FindBugs [11, 10] and of the work by Male *et al.* [15]. They rely on *path-sensitive* analysis and the treatment of field initialization is very weak.

To *infer type annotations*, Houdini [8] generates a set of possible annotations (non-null annotations among others) for a given program and uses ESC/Java [14] to refute false assumptions. CANAPA [3] lowers the burden of annotating a

program by propagating some non-null annotations. It also relies on ESC/Java to infer where annotations are needed. Those two annotation assistants have a simplified handling of objects under construction and are intented to be used by the developer to debug and not to certify programs. Indeed, they rely on ESC/Java [14], which is not sound (nor complete). JastAdd [5] is a tool to infer annotations for a simplified version of Fähndrich and Leino's type system.[4]

Finally, another approach to lower the amount of annotations is proposed by Chalin and James [2]. They suggest to consider the references as non-null by default, so the developer has only to explicitly annotate nullable references. Despite the lower amount of annotation needed, about 1/3 of declarations still need annotations which can represent a substantial amount of annotations in legacy code.

Contributions. The non-null reference analysis presented here makes the following contributions.

- The analysis is fully automatic so there is no annotation burden for the programmer.
- The soundness of the analysis is proved formally with respect to an operational semantics. This is the first formal correctness proof for this kind of analyses. Furthermore, this proof has been checked mechanically using the Coq proof assistant.
- We provide a detailed comparison with Fähndrich and Leino's type system, which is a reference among the nullness program analyses. The completeness with respect to their type system is proved. In this way, the correctness proof of our analysis also provides a formal proof of correctness of the type system of Fähndrich and Leino. We also show that our analysis can be slightly more precise than their type system.
- The analysis is modular: a program can be analysed without analyzing the libraries if they have already been annotated.

Outline. Section 2 presents the syntax and semantics of the simple OO language we use to formalize our analysis. Section 3 presents the system of constraints of the analysis and Sect. 4 gives the proof of soundness. Section 5 proves the constraint system has a least fixpoint and discusses the modularity. Section 6 then presents Fähndrich and Leino's type system and proves the completeness of our analysis with respect to their type system. Section 7 gives some details on the adaptation of the analysis to the Java bytecode level and Sect. 8 concludes.

[4] The treatment of objects under initialization is simplified and initializations done by methods (called from the constructor) are not taken in account.

2 Syntax and Semantics

We define a minimalistic language[5] to analyze the flow of references in object-based languages. A program is a collection of classes, arranged in a class hierarchy via a transitive subclass relation $\prec$ (we write $\preceq$ for its reflexive closure). We only consider single inheritance (where $\preceq$ can be embedded in a sup-semilattice structure). The language, as described in Fig. 1 has two kinds of expressions E: variables and references to fields. Assignments are either to variables or to fields. New objects are created using the **new** instruction which takes as arguments the name C of a class, the vector $\boldsymbol{\alpha}$ of the types of the parameters (used to deal with overloading) and a list of expressions. There is a conditional instruction which may non-deterministically branches to a given program label. Finally, the instructions $x.\mathrm{ms}(E,\ldots,E)$ and **return** represent method invocation and return, respectively. Methods invoked are found with a lookup procedure that looks for a method depending on its signature and the class of the current object. A method descriptor ms and a method identifier m are both of the form $\{\!|C, \mathrm{mn}, \boldsymbol{\alpha} \to \beta|\!\}$, where mn is a method name, $\boldsymbol{\alpha} \to \beta$ is a type signature (to simplify the presentation we here restrict β to **void**) and C is, for the method descriptor, the type of the reference on which the method is called and, for the method identifier, the class where the method has been defined. We use name(ms) and name(m) to retrieve mn and class(ms) and class(m) to retrieve C. A method signature is the couple $(\mathrm{mn}, \boldsymbol{\alpha} \to \beta)$. A method signature can correspond to many methods, a method descriptor corresponds to at most one method and a method identifier corresponds to only one method. As a notation abuse, m will be used as the method identified instead of the identifier itself. A method is composed of a method signature, a list of parameters and a method body consisting of a labelled list of instructions. A class is composed of fields and method definitions. For a given field f class(f) gives the class in which f is defined. The set of fields declared in a class C is written fields(C). Each program contains a special method **main** used to start the execution.

The operational semantics of our minimalistic language is defined by the inference rules in Fig. 1 which specify a (small-step) transition relation $\rightarrow_{\mathrm{m}}$ for each method m and a (big-step) evaluation relation $\Downarrow_{h,l}$ for expressions relative to a heap h and local variables l.

$$\Downarrow_{h,l} \subseteq E \times (\mathrm{Val} + \{\Omega\}) \qquad \rightarrow_{m} \subseteq \mathrm{State} \times (\mathrm{State} + \mathrm{Heap} + \{\Omega\})$$

The semantics uses an explicit error element Ω to signal the dereferencing of a null reference. Alternatively, we could have let the semantics "get stuck" when it encounters an error but our choice of propagating an error element to method boundaries facilitates the correctness proof of the null pointer analysis to follow. For space reasons, we only detail the error case for method invocation; all other instructions may lead to similar error steps. The analysis keeps track of

[5] The language is closed to the Java bytecode language but without any operand stack. Removing the operand stack avoid to introduce an alias analysis which is needed at the bytecode level, for instance, to know if a stack variable is an alias of **this**.

how objects are initialized and, in order to prove its correctness, the semantics instruments the fields of objects with flags **def** or **undef** to track the history of field initializations. A field flagged as **undef** contains necessarily null. This instrumentation is transparent in the sense that it does not affect the behavior of a program and will be used later to prove the correctness of the analysis. In Java bytecode, unlike fields, local variables do not have a default value and the bytecode verifier ensures uninitialized local variables are never read. We formalized this by using $\perp$ as the default value for local variables and by ensuring with the semantics $\perp$ is never read. We also model object initialization to ensure a constructor of class C terminates only when at least one constructor of each ancestor has been called. To do so, we keep for each object the set of classes whose constructor has been called. The semantics is stuck if a constructor violates that policy. This means we only consider programs where objects are correctly initialized.

Notation: We write $\mathbb{F}$, $\mathbb{V}$, Loc and $\mathbb{N}$ for fields, variables, memory addresses and program points, respectively. We consider that a field includes the class name where it has been defined. We note $h(r)(f)$, $\text{class}(h(r))$ and $\text{history}(h(r))$ the accesses to the first, second and third components of the heap cell $h(r)$, respectively. Function $\text{upd}(h, r, f, (v, \mathbf{def}))$ sets the field f of the object at location r to the value v and marks it as defined. The expression $\text{default}(C)$ denotes a new object of class C where all fields contain null values and are **undef** (the history of such object is empty). The function set2Def_C sets all fields (in $\text{fields}(C)$) of an object to **def**. The function addHistory_C adds a class name C to the initialization history of an object. The star transition $\rightarrow_{\text{m}}^*$ corresponds to the transitive closure of $\rightarrow_{\text{m}}$.

A program is considered *null-pointer error safe* if the execution of the method **main** never reaches an error state, starting from an empty heap with only one (uninitialised) object in it and all local variable (except **this**) with the default value $\perp$.

Definition 1. *A program P is said to be* null-pointer error safe *if for all location r,*

$$\langle 0, \perp[\boldsymbol{this} \mapsto r], h_0\rangle \rightarrow_{\boldsymbol{main}}^* s \text{ implies } s \neq \Omega$$

where h_0 is the heap of domain $\{r\}$ with $h_0(r) = \text{default}(\text{class}(\boldsymbol{main}))$.

Note that since error states are propagated after method calls, this definition of safe program implies that no dereferencing of a null reference will occur during the execution of the program.

3 Null-Pointer Analysis

We will now present the analysis that is able to prove a program is null-pointer error safe. As mentioned in the introduction, although the analysis annotates local variables, method parameters and return value, its purpose is to annotate the fields.

Syntax

$$
\begin{aligned}
E &::= x \mid E.F \\
I &::= x \leftarrow E \mid x.f \leftarrow E \mid x \leftarrow \mathbf{new}\ (C, \boldsymbol{\alpha})(E, \ldots, E) \mid \mathbf{if}\ (\star)\ \mathtt{jmp} \mid x.\mathrm{ms}(E, \ldots, E) \mid \mathbf{return} \\
M &::= \mathrm{mn}(x_1, \ldots, x_n)\ \{\ I; \ldots; I\ \} \\
C &::= \{\!|\mathsf{fields} : \{f; \ldots; f\}; \mathsf{methods} : \{M; \ldots; M\}|\!\} \\
P &::= (C\ \cdots\ C, \prec)
\end{aligned}
$$

Domains

$$
\begin{gathered}
\mathrm{Val} = \mathrm{Loc} + \{\mathbf{null}\} \qquad \mathrm{Object} = \mathbb{F} \rightharpoonup \mathrm{Val} \times \{\mathbf{def}, \mathbf{undef}\} \\
\mathrm{LocalVar} = \mathbb{V} \to \mathrm{Val} + \{\bot\} \qquad \mathrm{Heap} = \mathrm{Loc} \rightharpoonup \mathrm{Object} \times \mathbb{C} \times \wp(\mathbb{C}) \\
\mathrm{State} = (\mathbb{N} \times \mathrm{LocalVar} \times \mathrm{Heap}) + \Omega
\end{gathered}
$$

Expression Evaluation

$$
\frac{l(x) \neq \bot}{x \Downarrow_{h,l} l(x)} \qquad \frac{e \Downarrow_{h,l} r \qquad r \in \mathrm{dom}(h) \qquad f \in \mathrm{dom}(h(r))}{e.f \Downarrow_{h,l} h(r)(f)} \qquad \frac{e \Downarrow_{h,l} v \qquad v \in \{\mathbf{null}, \Omega\}}{e.f \Downarrow_{h,l} \Omega}
$$

Operational Semantics: Normal Cases

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x \leftarrow e} \qquad e \Downarrow_{h,l} v \qquad x \neq \mathbf{this}}{\langle i, l, h\rangle \to_{\mathrm{m}} \langle i+1, l[x \mapsto v], h\rangle} \qquad \frac{\boxed{P_{\mathrm{m}}[i] = x.f \leftarrow e} \qquad l(x) \in \mathrm{dom}(h) \qquad f \in \mathrm{dom}(h(l(x))) \qquad e \Downarrow_{h,l} v}{\langle i, l, h\rangle \to_{\mathrm{m}} \langle i+1, l, \mathrm{upd}(h, l(x), f, (v, \mathbf{def}))\rangle}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x \leftarrow \mathbf{new}\ (C, \boldsymbol{\alpha})(e_1, \ldots, e_n)} \qquad \forall i,\ e_i \Downarrow_{h,l} v_i \qquad r \notin \mathrm{dom}(h) \qquad x \neq \mathbf{this} \qquad \langle 0, \bot[\mathbf{this} \mapsto r,\ \{i \mapsto v_i\}_{i=1..n}], h[r \mapsto \mathrm{default}(C)]\rangle \to^*_{\{\!|C, \mathbf{init}, \boldsymbol{\alpha} \to \mathbf{void}|\!\}} h'}{\langle i, l, h\rangle \to_{\mathrm{m}} \langle i+1, l[x \mapsto r], h'\rangle}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x.\mathrm{ms}(e_1, \ldots, e_n)} \qquad \forall i,\ e_i \Downarrow_{h,l} v_i \qquad l(x) \in \mathrm{dom}(h) \qquad \mathrm{m}' = \mathrm{lookup}(class(h(l(x))), \mathrm{ms}) \qquad \mathrm{class}(h(l(x))) \preceq \mathrm{class}(\mathrm{ms}) \qquad \langle 0, \bot[\mathbf{this} \mapsto l(x),\ \{i \mapsto v_i\}_{i=1..n}], h\rangle \to^*_{\mathrm{m}'} h' \qquad \mathrm{name}(ms) = \mathbf{init} \Rightarrow x = \mathbf{this}}{\langle i, l, h\rangle \to_{\mathrm{m}} \langle i+1, l, h'\rangle}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = \mathbf{if}\ \mathrm{jmp}}}{\langle i, l, h\rangle \to_{\mathrm{m}} \langle \mathrm{jmp}, l, h\rangle} \qquad \frac{\boxed{P_{\mathrm{m}}[i] = \mathbf{if}\ \mathrm{jmp}}}{\langle i, l, h\rangle \to_{\mathrm{m}} \langle i+1, l, h\rangle}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = \mathbf{return}} \qquad C = \mathrm{class(m)} \qquad \mathrm{name(m)} = \mathbf{init} \Rightarrow \forall A, C \prec A \Rightarrow A \in \mathrm{history}(l(\mathbf{this})) \qquad \mathrm{name(m)} = \mathbf{init} \Rightarrow h' = h[l(\mathbf{this}) \mapsto \mathrm{addHistory}_C(\mathrm{set2Def}_C(h(l(\mathbf{this}))))] \qquad \mathrm{name(m)} \neq \mathbf{init} \Rightarrow h' = h}{\langle i, l, h\rangle \to_{\mathrm{m}} h'}
$$

Operational Semantics: Error Cases (Selected Rules)

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x.\mathrm{ms}(e_1, \ldots, e_n)} \qquad \forall i,\ e_i \Downarrow_{h,l} v_i \qquad l(x) \in \mathrm{dom}(h) \qquad \mathrm{m}' = \mathrm{lookup}(\mathrm{class}(h(l(x))), \mathrm{ms}) \qquad \mathrm{class}(h(l(x))) \preceq \mathrm{class}(\mathrm{ms}) \qquad \langle 0, \bot[\mathbf{this} \mapsto l(x),\ \{i \mapsto v_i\}_{i=1..n}], h\rangle \to^*_{\mathrm{m}'} \Omega \qquad \mathrm{name(m)} = \mathbf{init} \Rightarrow x = \mathbf{this}}{\langle i, l, h\rangle \to_{\mathrm{m}} \Omega}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x.\mathrm{ms}(e_1, \ldots, e_n)} \qquad l(x) = \mathbf{null} \vee \exists i,\ e_i \Downarrow_{h,l} \Omega}{\langle i, l, h\rangle \to_{\mathrm{m}} \Omega}
$$

Fig. 1. Syntax and semantics of the language

3.1 Abstract Domains

In this section we define an analysis that for each class of a well-typed program infers annotations about nullity of its fields together with pre- and post-conditions for its methods. The basic properties of interest are NotNull — meaning "definitely not-null" — and MayBeNull — meaning possibly a null reference. The field annotations computed by the analysis are represented as a heap abstraction $H^\sharp \in \text{Heap}^\sharp$ which provides an abstraction for all fields of all initialized objects and all initialized fields of objects being initialized.[6]

As explained in the Introduction, object initialization requires a special treatment because all fields are null when the object is created so this would lead a simple-minded analysis to annotate all fields as MayBeNull. However, we want to infer non-null annotations for all fields which are initialized explicitly to be so in a constructor. In other words, fields that are not initialized in a constructor are annotated as MayBeNull and all other fields have a type compatible with the value they have been initialized with.

To this end, the analysis tracks field initializations of the current object in constructors (and methods called from constructors). This is done via an abstraction of the **this** reference by a domain $\text{TVal}^\sharp$ which maps each of the fields declared in the current class to Def or UnDef. To allow strong updates, we need a flow-sensitive abstraction so to each program point we map a such abstraction ($T^\sharp$).

References are then abstracted by a domain $\text{Val}^\sharp$ which incorporate the "raw" references from [6]. A Raw^- value denotes a non-null reference of an object being initialized, which does not yet respect his invariant (*e.g.* Raw^- can be used as a property of `this` when it occurs in constructors). If a reference is known to have all its fields declared in class X and in the parents of X initialized, then the reference is $\text{Raw}(X)$. The inclusion of "raw" references allows the manipulation of objects during initialization because the analysis can use the fact that for an object of type $\text{Raw}(X)$, only fields declared in X and above have a valid annotation in the abstract heap $H^\sharp$. A NotNull value denotes a non-null reference that has finished its initialization.

Figure 2 defines formally each abstract domain. The flow of references through local variables is analysed with a flow-sensitive abstraction of each variable ($L^\sharp \in \text{LocalVar}^\sharp$). The analysis also infers method annotations $M^\sharp \in \text{Method}^\sharp$. For any method m, $M^\sharp(\text{m})[\mathsf{this}]$ is an approximation of the initialization state of **this** before the execution of m, while $M^\sharp(\text{m})[\mathsf{post}]$ gives the corresponding approximation at the end of the execution. $M^\sharp(\text{m})[\mathsf{args}]$ approximates the parameters of the method, taking into account all the context in which m may actually be invoked. We only give the definition of the partial order for $\text{Val}^\sharp$ and $\text{Def}^\sharp$. The other orders are defined in a standard way using the canonical orders on partial functions, products and lists. The final domain $\text{State}^\sharp$ is hence equipped with a straightforward lattice structure.

[6] We consider fields *initialized* when they have been assigned a value, whereas we consider an object *initialized* when it has returned from its constructor.

Abstract Domains

$\mathrm{Val}^\sharp = \{\mathrm{Raw}(Y) \mid Y \in \mathrm{Class}\} \cup \{Raw^-, \mathrm{NotNull}, \mathrm{MayBeNull}\}$			
$\mathrm{Def}^\sharp = \{\mathrm{Def}, \mathrm{UnDef}\}$	$\mathrm{TVal}^\sharp = \mathbb{F} \rightharpoonup \mathrm{Def}^\sharp$	$\mathrm{Heap}^\sharp = \mathbb{F} \to \mathrm{Val}^\sharp$	$\mathrm{LocalVar}^\sharp = \mathbb{V} \to \mathrm{Val}^\sharp$
$\mathrm{Method}^\sharp = \mathbb{M} \to \{\lvert \mathsf{this} \in \mathrm{TVal}^\sharp;\ \mathsf{args} \in (\mathrm{Val}^\sharp)^*; \mathsf{post} \in \mathrm{TVal}^\sharp \rvert\}$			
$\mathrm{State}^\sharp = \mathrm{Method}^\sharp \times \mathrm{Heap}^\sharp \times \left(\mathbb{M} \times \mathbb{N} \rightharpoonup \mathrm{TVal}^\sharp\right) \times \left(\mathbb{M} \times \mathbb{N} \rightharpoonup \mathrm{LocalVar}^\sharp\right)$			

Selected Partial Orders

$\mathrm{Val}^\sharp$		$\mathrm{Def}^\sharp$
$\dfrac{x \in \mathrm{Val}^\sharp}{\mathrm{NotNull} \sqsubseteq x}$	$\dfrac{}{\mathrm{Raw}(X) \sqsubseteq Raw^-}$	$\dfrac{}{\mathrm{Def} \sqsubseteq \mathrm{UnDef}}$
$\dfrac{X \preceq Y}{\mathrm{Raw}(X) \sqsubseteq \mathrm{Raw}(Y)}$	$\dfrac{x \in \mathrm{Val}^\sharp}{x \sqsubseteq \mathrm{MayBeNull}}$	

Fig. 2. Abstract domains and selected partial orders

3.2 Inference Rules

The analysis is specified via a set of inference rules, shown in Fig. 3, which define a judgment

$$M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m}, i) \ : \ instr$$

for when the abstract state $(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ is coherent with instruction *instr* at program point (m, i). For each such program point, this produces a set of constraints over the abstract domain $\mathrm{State}^\sharp$, whose solutions constitute the correct analysis results. The rules make use of an abstract evaluation function for expressions (explained below) that we write $[\![e]\!]^\sharp$. An example of a program with the corresponding constraints are given in Sect. 3.3.

Assignment to a local variable (rule (1)) simply assigns the abstract value associated with the expression to the local variable in the abstract environment $L^\sharp$. Assignment to a field (2) can either be to a field of the current object, in which case the field becomes "defined", or to another object. In both cases, the abstract heap $H^\sharp$ is augmented with the value of the expression as a possible value for the field. When a return value is encountered (3) in a constructor, all still-undefined fields are explicitly set to MayBeNull. For a `new` instruction (4), a new abstraction $\top_C$ is built for the pre-condition on `this` of the constructor. The first argument (`this`) is set to Raw^-. The result of the constructor is known to be NotNull (the object is fully initialized). The method call (6) uses conditional constraints to distinguish a number of cases, depending on whether the call is to a method in the current class and the receiving object is the current object `this` or not. We use $\mathrm{Raw}(\mathrm{super}(C))$ to denote the Raw type just above $\mathrm{Raw}(C)$.[7]

The whole constraint system of a program P is then formally defined by the judgement $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$. We write $\mathrm{overrides}(\mathrm{m}', \mathrm{m})$ when the method m' overrides m. In such case we require a contravariant property between the

[7] If C is the root of the class hierarchy (Object in Java), then $\mathrm{Raw}(\mathrm{super}(C)) = Raw^-$

Inference Rules

$$\frac{T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1) \qquad L^\sharp(\mathrm{m},i)[x \mapsto [\![e]\!]^\sharp] \sqsubseteq L^\sharp(\mathrm{m},i+1)}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ x \leftarrow e} \tag{1}$$

$$\frac{\begin{array}{c}\text{if } x = \textbf{this} \wedge f \in \text{fields}(\text{class}(\mathrm{m})) \text{ then } T^\sharp(\mathrm{m},i)[f \mapsto \text{Def}] \text{ else } T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1) \\ L^\sharp(\mathrm{m},i) \sqsubseteq L^\sharp(\mathrm{m},i+1) \qquad [\![e]\!]^\sharp \sqsubseteq H^\sharp(f)\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ x.f \leftarrow e} \tag{2}$$

$$\frac{\begin{array}{c}T^\sharp(\mathrm{m},i) \sqsubseteq M^\sharp(\mathrm{m})[\mathsf{post}] \\ \text{name}(\mathrm{m}) = \textbf{init} \Rightarrow \forall f \in \text{fields}(\text{class}(\mathrm{m})).(T^\sharp(\mathrm{m},i)(f) = \text{UnDef} \Rightarrow \text{MayBeNull} \sqsubseteq H^\sharp(f))\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ \textbf{return}} \tag{3}$$

$$\frac{\begin{array}{c}\mathrm{m}' = \{\!|C, \textbf{init}, \boldsymbol{\alpha} \to \textbf{void}|\!\} \\ \top_C \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{this}] \qquad Raw^- :: [\![e_1]\!]^\sharp :: \cdots :: [\![e_j]\!]^\sharp \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{args}] \\ L^\sharp(\mathrm{m},i)[x \mapsto \text{NotNull}] \sqsubseteq L^\sharp(\mathrm{m},i+1) \qquad T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1)\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ x \leftarrow \textbf{new } (C, \boldsymbol{\alpha})(e_1, \ldots, e_j)} \tag{4}$$

$$\frac{\begin{array}{c}L^\sharp(\mathrm{m},i) \sqsubseteq L^\sharp(\mathrm{m},i+1) \qquad L^\sharp(\mathrm{m},i) \sqsubseteq L^\sharp(\mathrm{m},\text{jmp}) \\ T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1) \qquad T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},\text{jmp})\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ \textbf{if jmp}} \tag{5}$$

$$\frac{\begin{array}{c}\mathrm{m}' = \text{lookup}(\text{class}(\text{ms}), \text{ms}) \qquad C' = \text{class}(\mathrm{m}') \qquad C = \text{class}(\mathrm{m}) \\ \left(\begin{array}{l}\text{if } x = \textbf{this} \wedge C = C' \wedge [\![x]\!]^\sharp \sqsubseteq \text{Raw}(\text{super}(C')) \wedge M^\sharp(\mathrm{m}')[\mathsf{post}] \sqcap T^\sharp(\mathrm{m},i) \sqsubseteq \bot_{C'} \\ \text{then } L^\sharp(\mathrm{m},i)[x \mapsto \text{Raw}(C') \sqcap [\![x]\!]^\sharp] \sqsubseteq L^\sharp(\mathrm{m},i+1) \\ \text{else if name}(\mathrm{m}') = \textbf{init} \text{ then } L^\sharp(\mathrm{m},i)[x \mapsto \text{Raw}(C')] \sqsubseteq L^\sharp(\mathrm{m},i+1) \\ \text{else } L^\sharp(\mathrm{m},i) \sqsubseteq L^\sharp(\mathrm{m},i+1)\end{array}\right) \\ \left(\begin{array}{l}\text{if } x = \textbf{this} \wedge C = C' \text{ then } M^\sharp(\mathrm{m}')[\mathsf{post}] \sqcap T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1) \\ \text{else } T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1)\end{array}\right) \\ \left(\begin{array}{l}\text{if } x = \textbf{this} \wedge C = C' \text{ then } T^\sharp(\mathrm{m},i) \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{this}] \\ \text{else } \rho(C', [\![x]\!]^\sharp) \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{this}]\end{array}\right) \\ [\![x]\!]^\sharp :: [\![e_1]\!]^\sharp :: \cdots :: [\![e_j]\!]^\sharp \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{args}]\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ x.\text{ms}(e_1, \ldots, e_j)} \tag{6}$$

$$\frac{\begin{array}{c}\forall \mathrm{m}, \mathrm{m}', \text{overrides}(\mathrm{m}', \mathrm{m}) \Rightarrow M^\sharp(\mathrm{m})[\mathsf{args}] \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{args}] \\ \forall \mathrm{m}, \mathrm{m}', \text{overrides}(\mathrm{m}', \mathrm{m}) \Rightarrow \top_{\text{class}(\mathrm{m})} \sqsubseteq M^\sharp(\mathrm{m})[\mathsf{post}] \\ \forall \mathrm{m}, \mathrm{m}', \text{overrides}(\mathrm{m}', \mathrm{m}) \Rightarrow \top_{\text{class}(\mathrm{m}')} \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{this}] \\ \forall \mathrm{m}, \ M^\sharp(\mathrm{m})[\mathsf{this}] \sqsubseteq T^\sharp(\mathrm{m},0) \qquad \forall \mathrm{m}, \ M^\sharp(\mathrm{m})[\mathsf{args}] \sqsubseteq L^\sharp(\mathrm{m},0) \\ \top_{\text{class}(\textbf{main})} \sqsubseteq T^\sharp(\textbf{main},0) \qquad Raw^- \sqsubseteq L^\sharp(\textbf{main},0)(\textbf{this}) \\ \forall \mathrm{m}, \ \forall i, \ M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ P_{\mathrm{m}}[i]\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P} \tag{7}$$

Auxiliary Operators

$$\begin{aligned}\rho(C, \text{NotNull}) &= \bot_C \\ \rho(C, \text{Raw}(X)) &= \text{if } X \preceq C \text{ then } \bot_C \text{ else } \top_C \\ \rho(C, \text{MayBeNull}) &= \rho(C, Raw^-) = \top_C\end{aligned}$$

$$[\![x]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp} = \begin{cases}\text{Raw}(C) & \text{if } x = \textbf{this} \wedge t^\sharp = \bot_c \wedge l^\sharp(x) = \text{Raw}(\text{super}(C)) \\ L^\sharp(x) & \text{otherwise}\end{cases}$$

$$[\![e.f]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp} = \begin{cases}H^\sharp(f) & \text{if } [\![e]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp} = \text{NotNull} \\ & \text{or } (e = \textbf{this} \wedge t^\sharp(f) = \text{Def} \wedge \text{class}(f) = C) \\ & \text{or } [\![e]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp} = \text{Raw}(X) \text{ with } X \preceq \text{class}(f) \\ \text{MayBeNull} & \text{otherwise}\end{cases}$$

Fig. 3. Analysis specification

parameters of m and m′. Any overriding overrides(m′, m) need also to invalidate (in term of precision) $M^\sharp(\mathrm{m})[\mathsf{post}]$ because a virtual call to m could lead to the execution of m′ which is not able (by definition of $T^\sharp$) to track the initialization of fields declared in class(m). A similar constraint is required for $M^\sharp(\mathrm{m}')[\mathsf{this}]$ because for a virtual call to m we have only constrained $M^\sharp(\mathrm{m})[\mathsf{this}]$ and not $M^\sharp(\mathrm{m}')[\mathsf{this}]$. The constraints on $T^\sharp(\mathbf{main}, 0)$ and $L^\sharp(\mathbf{main}, 0)(\mathbf{this})$ ensure a correct initialisation of the **main** method.

Finally an abstract state is said safe (noted $\mathrm{safe}^\sharp_P(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$) when for each program point (m, i), if $P_\mathrm{m}[i]$ is of the form $x.f \leftarrow e$ or $x.\mathrm{ms}(\ldots)$ then $L^\sharp(\mathrm{m}, i)(x) \neq \mathrm{MayBeNull}$, and for all expressions e which appear in the instruction $P_\mathrm{m}[i]$, any dereferenced sub-expression e' has an abstract evaluation strictly lower than MayBeNull.

The analysis uses a function ρ to transfer information from domain $\mathrm{Val}^\sharp$ to domain $\mathrm{TVal}^\sharp$: ρ transforms an abstract reference $\mathrm{Val}^\sharp$ to a $\mathrm{TVal}^\sharp$ abstraction of a current object `this` in the class C. The notations $\bot_C$, respectively $\top_C$, correspond to the function that maps all fields defined in C to Def, respectively UnDef. The analysis also relies on an abstract evaluation $[\![e]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp}$ of expressions parameterised by the current class C, an abstract heap $H^\sharp$, an abstraction $t^\sharp$ of the fields (declared in class C) of the **this** object and an abstraction $l^\sharp$ of the local variables. The first equation states that the type of a local variable is obtained from the $L^\sharp$ function and can be refined from $\mathrm{Raw}(\mathrm{super}(C))$ to $\mathrm{Raw}(C)$ if **this** is sufficiently initialized. The second equation states that the type of a field is obtained from the heap abstraction if the type of the reference is NotNull or if it is a reference to an object sufficiently deeply initialized. Otherwise, it is MayBeNull. In the inference rules we write $[\![e]\!]^\sharp$ for $[\![e]\!]^\sharp_{\mathrm{class(m)},H^\sharp,T^\sharp(\mathrm{m},i),L^\sharp(\mathrm{m},i)}$

3.3 Example

The Java source code of our example is provided in Fig. 4, while Fig. 5 shows the code in the syntax defined in Sect. 2 and the constraints obtained from the rules defined in Sect. 3.2. This example is fairly simple and, for conciseness, the code of the `main` method has been omitted, but the generated constraints should be sufficient to give an idea of how the inference works.

The labels in the source starting with the character `@` are the annotations inferred by the analysis for the fields and the methods signatures. An annotation in front of a method corresponds to the property of **this** before the invocations of the method. The other annotations are placed just before the variable they refer to.

Lines 6, 7, 24 and 25 list the constraints obtained from rule (7) in Fig. 3. They "initialize" the flow-sensitive abstractions $L^\sharp(\mathrm{m}, 0)$ and $T^\sharp(\mathrm{m}, 0)$ with the information of the method signatures. All other constraints are directly deduced from the rule corresponding to the instruction where conditional constraints have been simplified where it was possible. Lines 9 to 12 list the constraints obtained from rule (6) when $\mathrm{name}(\mathrm{m}') = \mathbf{init}$ and $\mathrm{class}(\mathrm{m}) \neq \mathrm{class}(\mathrm{m}')$. Lines 14 to 16 correspond to a field assignment on **this** of a field defined in the current class.

```
class A {
    private Object f;
    private Object g;

    A(Object o){f = o;}

    void m(){g=f;}

    public static void main(String args[]){
        Object o = new Object();
        A a = new A(o);
        a.m();
    }
}
```

Fig. 4. Java source

Lines 18 to 20 correspond to a **return** instruction of a constructor, which adds the value MayBeNull in the abstract heap for all (maybe) undefined fields, while line 31 corresponds to a **return** instruction of a non-constructor method.

The methods are called from the `main` method of Fig. 4. The method `init` is annotated with Raw^- (`@RawTop`) as it is called on a completely uninitialized value. The constraint line 11 then refine Raw^- to Raw(*Object*) so if a field of `Object` were accessed in the rest of the method, the abstraction of the heap would be used. The NotNull (`@NotNull`) annotation before the argument of the method `init` is first transferred to a local variable at line 6 and then moved from o to $H^\sharp$(A.f) at line 14. At the same time, line 15 records that the field `f` is defined. Then, line 18, as `f` has been defined $H^\sharp$(A.f) is not modified whereas, line 19, $T^\sharp(\text{A.init}, 2)(\text{A.g}) = \text{UnDef}$ so $H^\sharp$(A.g) is constrained to MayBeNull.

4 Correctness

In this section we prove the correctness of the analysis. We first define (see Fig. 6) the logical link between concrete and abstract domains via a correctness relation. Then we prove (Theorem 1) that any solution of the constraint system which verifies the predicate $\text{safe}_p^\sharp$ enforces the null-pointer error safety property for the set of preconditions inferred by the analysis. The result mainly relies on a suitable subject reduction lemma (Lemma 1). The theorem has been mechanically proved with the Coq proof assistant[8].

[8] The proof is available at http://www.irisa.fr/lande/pichardie/np/

```
 1: class A {
 2:    Object @NotNull f;
 3:    Object @MayBeNull g;
 4:
 5:    @RawTop void init(Object @NotNull o){
 6:    M♯(A.init)[args] ⊑ L♯(A.init, 0)
 7:    M♯(A.init)[this] ⊑ T♯(A.init, 0)
 8:       0: this.{|Object,init,[] -> void|}()
 9:          ρ(Object, [[this]]) ⊑ M♯(Object.init)[this]
10:          [] ⊑ M♯(Object.init)[args]
11:          L♯(A.init, 0)[this ↦ Raw(Object)] ⊑ L♯(A.init, 1)
12:          T♯(A.init, 0) ⊑ T♯(A.init, 1)
13:       1: this.f = o
14:          [[o]] ⊑ H♯(A.f)
15:          T♯(A.init, 1)[A.f ↦ Def] ⊑ T♯(A.init, 2)
16:          L♯(A.init, 1) ⊑ L♯(A.init, 2)
17:       2: return
18:          if T♯(A.init, 2)(A.f) = UnDef then MayBeNull ⊑ H♯(A.f)
19:          if T♯(A.init, 2)(A.g) = UnDef then MayBeNull ⊑ H♯(A.g)
20:          T♯(A.init, 2) ⊑ M♯(A.init)[post]
21:    }
22:
23:    @NotNull void m(){
24:    M♯(A.m)[args] ⊑ L♯(A.m, 0)
25:    M♯(A.m)[this] ⊑ T♯(A.m, 0)
26:       0: this.g = this.f
27:          [[this.f]] ⊑ H♯(A.g)
28:          T♯(A.m, 0)[A.g ↦ Def] ⊑ T♯(A.m, 1)
29:          L♯(A.m, 0) ⊑ L♯(A.m, 1)
30:       1: return
31:          T♯(A.m, 1) ⊑ M♯(A.m)[post]
     }

  }
```

Fig. 5. Code with constraints

$$\frac{v \in \mathrm{Val}}{\mathrm{Def} \sim (v, \mathbf{def})} \qquad \frac{v \in \mathrm{Val} \quad d \in \{\mathbf{def}, \mathbf{undef}\}}{\mathrm{UnDef} \sim (v, d)}$$

$$\frac{v \in \mathrm{dom}(h) \quad \forall f \in \mathrm{dom}(h(v)), \mathrm{IsDef}(h(v)(f))}{\mathrm{NotNull} \sim_h v} \qquad \frac{v \in \mathrm{Val}}{\mathrm{MayBeNull} \sim_h v}$$

$$\frac{v \in \mathrm{dom}(h) \quad \forall f \in \bigcup_{A \preceq C} \mathrm{fields}(C) \cap \mathrm{dom}(h(v)), \mathrm{IsDef}(h(v)(f))}{\mathrm{Raw}(A) \sim_h v} \qquad \frac{v \neq \mathbf{null}}{Raw^- \sim_h v}$$

$$\frac{\forall x, l(x) = \bot \vee L^\sharp(x) \sim_h l(x)}{L^\sharp \sim_h l} \qquad \frac{\begin{array}{c} r \in \mathrm{dom}(h) \quad \mathrm{fields}(C) \subseteq \mathrm{dom}(T^\sharp) \cap \mathrm{dom}(h(r)) \\ \forall f \in \mathrm{fields}(C),\ T^\sharp(f) \sim h(r)(f) \end{array}}{T^\sharp \sim_{h,C} r}$$

$$\frac{\forall r \in \mathrm{dom}(h),\ \forall f \in \mathrm{dom}(h(r)),\ h(r)(f) = (v, d) \Rightarrow d = \mathbf{undef} \vee H^\sharp(f) \sim_h v}{H^\sharp \sim h}$$

$$\frac{o = l(\mathbf{this}) \quad L^\sharp(\mathrm{m}, i) \sim_h l \quad T^\sharp(\mathrm{m}, i) \sim_{h,\mathrm{class(m)}} l(\mathbf{this}) \quad H^\sharp \sim h}{(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{\mathrm{m},o} \langle i, l, h \rangle}$$

$$\frac{M^\sharp(\mathrm{m})[\mathsf{post}] \sim_{h,\mathrm{class(m)}} o \quad H^\sharp \sim h}{(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{\mathrm{m},o} h} \qquad \frac{}{(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{\mathrm{m},o} \Omega}$$

Fig. 6. Correctness relations

An abstract element $(M^\sharp, H^\sharp)$ induces a set of method preconditions defined by

$$\mathrm{Pre}(M^\sharp, H^\sharp)(\mathrm{m}) = \left\{ (l, h) \;\middle|\; \begin{array}{l} H^\sharp \sim h,\ M^\sharp(\mathrm{m})[\mathsf{this}] \sim_{\mathrm{class(m)},h} l(\mathbf{this}), \\ V_0^\sharp \sim_h l(\mathbf{this}) \text{ and } \forall i = 1..n, V_i^\sharp \sim_h l(i) \\ \text{with } M^\sharp(\mathrm{m})[\mathsf{args}] = V_0^\sharp :: \cdots :: V_n^\sharp \end{array} \right\}$$

Note that a method m which is not called by any other method in the program, will be associated with a non-null assumption on all its parameters. This corresponds to the *non-null by default* approach advocated by Chalin and James [2].

We write $\rightarrow_{\mathrm{m}}^{(n)}$ (resp $\rightarrow_{\mathrm{m}}^{(n)*}$) for the operational step relation (resp. transitive closure) where exactly n method calls are performed during step (resp. transitive closure), including sub-calls.

Lemma 1 (Subject reduction). *Let $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \in State^\sharp$ such that $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ and $safe_P^\sharp(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$. Let n be an integer such that for all k, $k < n$, and all methods m, local variables l, heap h and configuration X if $(l, h) \in Pre(M^\sharp, H^\sharp)(m)$ and $\langle 0, l, h \rangle \rightarrow_m^{(k)*} X$ then $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{m,l(\boldsymbol{this})} X$ and $X \neq \Omega$.*

Let p be an integer, i a program counter, l local variables, h a heap and X a configuration such that $\langle i, l, h \rangle \rightarrow_m^{(p)} X$, $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{m,o} \langle i, l, h \rangle$ and $p \leq n$ then $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{m,o} X$ and $X \neq \Omega$.

Proof. See Appendix C in [12]

Theorem 1 (constraint system soundness). *If there exists $(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ such that $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ and $safe_P^\sharp(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ holds then P is null-pointer error safe.*

Proof. The proof proceeds by induction on the maximum number of method calls in the execution sequences and then another induction on the length of intra-method derivation, in order to be able to conclude with the subject reduction Lemma 1.

5 Inference

Expressing the analysis in terms of constraints over lattices has the immediate advantage that inference can be obtained from standard iterative constraint solving techniques for static analyses. Proposition 1 asserts that there is a decision procedure to detect programs which are verifiable with the analysis.

Proposition 1. *For all program P there exists an algorithm which decides if there exists $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \in State^\sharp$ such that $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ and $safe^\sharp_P(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$.*

Proof. It is a standard proof since the system of constraint is composed of monotone functions on a finite lattice. The least solution can then be computed and be checked with respect to predicate $\text{safe}^\sharp_P$.

Furthermore, the present analysis is modular in the sense that, rather than performing an analysis of all classes of a program, it is possible to describe certain classes (*e.g.*, classes coming from a library) by providing *interfaces* consisting of some method signatures $M^\sharp(\mathrm{m})$, $\mathrm{m} \in M^{\text{fix}}$ and field invariants $H^\sharp(f)$, $f \in F^{\text{fix}}$ relative to the classes.[9] The validity of these signatures can be established once and for all by the modular type checker proposed by Fähndrich and Leino (which works class by class). The analysis of partial programs would then proceed in several stages. First, the constraint system is generated only for the available classes. The system may refer to the variables $M^\sharp(\mathrm{m})$, $\mathrm{m} \in M^{\text{fix}}$ and $H^\sharp(f)$, $f \in F^{\text{fix}}$. Then the partial system is solved starting the iteration *à la* Kleene from

$$(M^\sharp_0, H^\sharp_0, T^\sharp_0, L^\sharp_0) = (\bot \sqcup \{\mathrm{m} \mapsto M^\sharp(\mathrm{m})\}_{\mathrm{m} \in M^{\text{fix}}}, \bot \sqcup \{f \mapsto H^\sharp(f)\}_{f \in F^{\text{fix}}}, \bot, \bot)$$

If one of the variables $M^\sharp(\mathrm{m})$, $\mathrm{m} \in M^{\text{fix}}$ or $H^\sharp(f)$, $f \in F^{\text{fix}}$ has to be updated during the iteration, the constraint resolution fails, since this means there does not exist a solution compatible with the proposed signatures.

Theorem 2 (Relative completeness of the modular solver). *If the previous algorithm halts then the partial constraint system has no solution compatible with the signatures provided for the unknown classes.*

Proof. Let S denote the set

$$\{(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \mid M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P \wedge (M^\sharp_0, H^\sharp_0, T^\sharp_0, L^\sharp_0) \sqsubseteq (M^\sharp, H^\sharp, T^\sharp, L^\sharp)\} .$$

[9] Modular inference is less precise: to keep the same precision the analysis would need richer annotations for the libraries.

Suppose the previous algorithm halts. Since the iteration is ascending it means the least solution of S is necessarily strictly greater than $(M_0^\sharp, H_0^\sharp, T_0^\sharp, L_0^\sharp)$. Since any solution compatible with the signatures $M^\sharp(\mathrm{m})$, $\mathrm{m} \in M^{\mathrm{fix}}$ or $H^\sharp(f)$, $f \in F^{\mathrm{fix}}$ is in S, we can conclude that such a solution does not exist.

6 Fähndrich and Leino's Type System

In this section we compare our analysis with the type system proposed by Fähndrich and Leino [6]. This comparison relies on a formal definition[10] of their notion of typable program. For conciseness, the full type system is not included here but can be found in [12].

A typing judgment for expression e is of the form $\Gamma, L \vdash e : \tau$ with Γ a type annotation for fields and methods, L a type annotation for local variables and τ a type in $\mathrm{Val}^\sharp$. Each instruction instr is also associated to a type judgement $\Gamma, \mathrm{m} \vdash \mathrm{instr} : L \to L'$ where m is the current method, L the current annotation for local variables and L' a valid annotation after the execution of instr. [`inits` F] is a method annotation that indicates the method initializes the fields in the set F. A program is said *well-typed* w.r.t. Γ if there are no overridden methods annotated as [`inits` F], arguments are contravariant and there exists L such that for all methods m and for all program points i, either $P_{\mathrm{m}}[i] = \textbf{return}$ or for all successors points j of i, their exists L' such that $L' \sqsubseteq L(\mathrm{m}, j)$ and $\Gamma, \mathrm{m} \vdash P_{\mathrm{m}}[i] \ : \ L(\mathrm{m}, i) \to L'$.

Figure 7 shows, as an example, the judgment for field assignments. It checks two properties: 1) the type $\Gamma[f]$ of the field f subsumes the type τ of the expression e and 2) the type of the reference x is not MayBeNull.

$$\frac{\Gamma, L \vdash e : \tau \qquad \tau \sqsubseteq \Gamma[f] \qquad L(x) \neq \mathrm{MayBeNull}}{\Gamma, \mathrm{m} \vdash x.f \leftarrow e \ : \ L \to L}$$

Fig. 7. Typing judgment for field assignments

This type system is coupled with a data flow analysis to ensure that all fields not declared as MayBeNull in class C are sure to be defined at the end of all constructors of C. It is a standard data flow analysis, the full description can be found in [12]. $\mathrm{F}(\mathrm{m}, i)$ represents the fields that are not initialized at program point (m, i). At the beginning of every constructor m, it is constrained to the set of fields defined in the class of m. For field assignation on **this**, the field is not propagated to the next node. For method calls to methods tagged as [`inits` F], fields in F are not propagated to the next node, but the set of undefined fields is propagated to the first instruction of the callee. It is then checked that those tagged methods initialize their fields.

[10] Note that in their paper the authors only propose an informal definition so what we formalise here is only our interpretation of their work.

Figure 8 shows an extract of the data flow analysis. It distinguishes two cases depending on whether the field assignation is on **this**, in which case the field assigned is removed from the set of undefined fields, or on another object, in which case the set of undefined fields is propagated.

$$\frac{\mathrm{F}(\mathrm{m},i) \setminus \{f\} \subseteq \mathrm{F}(\mathrm{m},i+1)}{\Gamma, \mathrm{F} \models (\mathrm{m},i) : \mathbf{this}.f \leftarrow e} \qquad \frac{x \neq \mathbf{this} \quad \mathrm{F}(\mathrm{m},i) \subseteq \mathrm{F}(\mathrm{m},i+1)}{\Gamma, \mathrm{F} \models (\mathrm{m},i) : x.f \leftarrow e}$$

Fig. 8. Data flow rule for field assignments

Theorem 3. *If P is FL-typable then there exists $(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ such that $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ and $safe^\sharp_P(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ holds.*

Proof. We show that if P is FL-typable for a given Γ and L then this type annotations represent a valid solution of the constraint system which furthermore satisfies the $\mathrm{safe}^\sharp_P$ property.

Corollary 1 (FL type system soundness). *If there exists Γ such that P is FL-typable then P is null-pointer error safe w.r.t. the preconditions given by Γ.*

Proof. Direct consequence of Theorem 3 and Theorem 1.

Theorem 4. *There exists P such that P is not FL-typable and there exists $(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ such that $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ and $safe^\sharp_P(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ holds.*

Proof. As shown in Fig. 3, the analysis of expressions is parametrized on the abstraction of `this` in order to know if a field has already been defined or not. In Fähndrich and Leino's analysis, the type checking is separated from the data-flow analysis that knows which fields have already been defined. For example, in

```
class C {
  Object f; //NotNull
  Object g; //MayBeNull or NotNull?
  public C(){ this.f = new Object(); this.g = this.f;}
}
```

our analysis benefits from the abstraction of `this` and knows `this.f` has been initialized before it is assigned to `this.g`. In Fähndrich and Leino's analysis, an intermediate local variable set to the new object and affected to `f` and `g` or an explicit cast operator which checks the initialization at run-time would be needed in order to type the program. Our abstraction of `this` can also be passed from a method to another, here also keeping some more information as the intra-procedural data-flow of Fähndrich and Leino. In the Soot suite we have analyzed [16] (see the next section), 5% of the constructors use fields that have been just defined.

7 Towards a Null-Pointer Analyzer for Java Bytecode

A prototype of the analysis has been implemented. The analysis works at the Java bytecode (JVML) level but annotations on fields and method can be propagated at the source level with a minor effort.

Working in the fragment of JVML without exceptions does not modify the analysis but some points need to be taken in account. JVML is a **stack language**, we therefore need to track aliases of `this` on the stack to know when a field manipulation or a method call is actually done on `this`. Operations on **numbers** do not add any difficulty as JVML is typed and numbers are guarantied not to be assigned to references. The **multi-threading** cannot interfere with weak updates, but it could interfere with the strong updates used on the abstraction of `this`. However, as the initialization evolves monotonically (*i.e.* a field once defined cannot be reverted to undefined) it is still correct. **Static methods** do not have an abstraction of `this` and do not add any other difficulty to handle.

Static fields are difficult to analyse precisely as they are initialized through `<clinit>` methods. **Exceptions** in Java can be Raw object, *i.e.* it is possible in a constructor to throw **this** as an exception. We could imagine a solution where abstract method signatures would include the type of the exceptions a method can throw. Tracking initialization of **array** cells precisely is also challenging as checking that all cells have been initialized requires numerical abstraction. We currently and conservatively annotate static fields and array cells as MayBe-Null and references to caught exception objects as Raw^- and left their precise handling for future work.

Other small optimizations (that have not been formalized) are possible. For example, the abstraction of `this` could be done for all other references of type $\mathrm{Raw}(X)$ (or Raw^-). It would give a more precise information on fields in general.

The first performance tests are promising as large programs such as Javacc 4.0 and Soot 2.2.4 can be analysed on a laptop in about 40 seconds and 5 minutes respectively. The analysis still needs to be completed with a path-sensitive analysis [4] to recover non-nullness information from the conditionals.

8 Conclusions and Future Work

We have defined a semantics-based analysis and inference technique for automatically inferring non-null annotations for fields. The analysis has been proved correct and the correctness proof has been machine-checked in the proof assistant Coq. This extends and complements the seminal paper of Fähndrich and Leino in which is proposed an extended type system for verifying non-null type annotations. Fähndrich and Leino's approach mixes type system and data-flow analysis. In our work, we follow an abstract interpretation methodology to gain strong semantic foundations and a goal-directed inference mechanism to find a minimal (*i.e.* principal) non-null annotation. By the same token we also gained in precision thanks to a better communication between abstract domains. We

then proved the correctness of our analysis and its completeness with respect to their type system.

Variations of the present analysis can be envisaged. For example, in our analysis, preconditions for methods are computed as the least upper bounds of the conditions verified at call points. Another approach would be to infer the weakest preconditions that prevents null-pointer exceptions.

In the future, we also plan to extend our analysis and its correctness proof to the full Java bytecode language. To manage this substantial extension it is important to be able to machine-check the correctness proof, which will necessarily be large. Our previous experience with developing a certified static analyser for Java [1] using the Coq proof assistant leads us to believe that such a formalisation is indeed feasible.

References

1. D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.
2. P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proc. of European Conference on Object-Oriented Programming (ECOOP'07)*, pages 227–247. Springer, 2007.
3. M. Cielecki, J. Fulara, K. Jakubczyk, and Ł. Jancewicz. Propagation of JML non-null annotations in Java programs. In *Proc. of the 4th international symposium on Principles and practice of programming in Java (PPPJ'06)*, pages 135–140. ACM Press, 2006.
4. M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proc. of the Conference on Programming language design and implementation (PLDI'02)*, pages 57–68. ACM, 2002.
5. T. Ekman and G. Hedin. Pluggable non-null types for Java. In Torbjörn Ekman, editor, *Extensible Compiler Construction*, chapter V. Lund University, June 2006.
6. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, pages 302–312. Springer-Verlag, 2003.
7. M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA'07: Proc. of the 22nd conference on Object Oriented Programming Systems and Applications*, pages 337–350. ACM, 2007.
8. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. of International Symposium of Formal Methods Europe (FME'01)*, pages 500–517. Springer-Verlag, 2001.
9. S. N. Freund and J. C. Mitchell. A formal framework for the java bytecode language and verifier. In *Proc. of the 14th conference on Object-oriented programming, systems, languages, and applications (OOPSLA'99)*, pages 147–166. ACM Press, 1999.
10. D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE'07: Proc. of the 7th workshop on Program analysis for software tools and engineering*, pages 9–14. ACM Press, 2007.
11. D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Softw. Eng. Notes*, 31(1):13–19, 2006.

12. L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. Research Report 6482, INRIA, March 2008.
13. M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware trap. *SIGPLAN Not.*, 35(11):139–149, 2000.
14. K. R. M. Leino, J. B. Saxe, and R. Stata. *ESC/Java user's manual*. Compaq Systems Research Center, technical note 2000-002 edition, October 2000.
15. Chris Male, David J. Pearce, Alex Potanin, and Constantine Dymnikov. Java bytecode verification for @NonNull types. In *Proc. of the Conference on Compiler Construction (CC'08)*. Springer-Verlag, 2008.
16. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot — a Java bytecode optimization framework. In *CASCON'99: Proc. of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.