Two techniques to improve the precision of a demand-driven null-dereference verification approach

Amogh Margoor[a], Raghavan Komondoor[a,*]

[a]*Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India, Phone: +91-80-22932368.*

Abstract

The problem addressed in this paper is sound, scalable, demand-driven null-dereference verification for Java programs. Our approach consists conceptually of a base analysis, plus two major extensions for enhanced precision. The base analysis is a dataflow analysis wherein we propagate formulas in the backward direction from a given dereference, and compute a necessary condition at the entry of the program for the dereference to be potentially unsafe. The extensions are motivated by the presence of certain "difficult" constructs in real programs, e.g., virtual calls with too many candidate targets, and library method calls, which happen to need excessive analysis time to be analyzed fully. The base analysis is hence configured to skip such a difficult construct when it is encountered by dropping all information that has been tracked so far that could potentially be affected by the construct. Our extensions are essentially more precise ways to account for the effect of these constructs on information that is being tracked, without requiring full analysis of these constructs. The first extension is a novel scheme to transmit formulas along certain kinds of def-use edges, while the second extension is based on using manually constructed backward-direction summary functions of library methods. We have implemented our approach, and applied it on a set of real-life benchmarks. The base analysis is on average able to declare about 84% of dereferences in each benchmark as safe, while the two extensions push this number up to 91%.

Keywords: dataflow analysis, weakest pre-conditions

1. Introduction

Null-dereferences are a bane while programming in pointer-based languages such as C and Java. In this paper, we describe a *sound*, *context-sensitive*, *demand-driven* technique to verify dereferences in Java programs via over-

*Corresponding author

Email addresses: `amogh.margoor@csa.iisc.ernet.in` (Amogh Margoor), `raghavan@csa.iisc.ernet.in` (Raghavan Komondoor)

```
1: foo(a,b,c) {
2:  if(a ≠ null) { $\langle \underline{\texttt{b.f} = null}, \texttt{a} \neq null, \texttt{a} = null \rangle \equiv false$
3:    b = c;         $false$
4:    t = new...;    $\langle \underline{\texttt{b.f} = null}, \texttt{b} \neq \texttt{c}, \texttt{a} \neq null \rangle$
5:    c.f = t;       $\langle \underline{\texttt{b.f} = null}, \texttt{b} \neq \texttt{c}, \texttt{a} \neq null \rangle, \langle \underline{\texttt{t} = null}, \texttt{b} = \texttt{c}, \texttt{a} \neq null \rangle$
6:  }
7:  d=a;             $\langle \underline{\texttt{b.f} = null}, \texttt{a} \neq null \rangle$
8:  if(d ≠ null)     $\langle \underline{\texttt{b.f} = null}, \texttt{d} \neq null \rangle$
9:    b.f.g = 10;    $\langle \underline{\texttt{b.f} = null} \rangle$
10:}
```

Figure 1: Example to illustrate the base analysis. The dereference of field `f` in line 9 is being verified.

approximated weakest pre-conditions analysis. A *weakest pre-condition wp(p,C)* is the weakest constraint on the *initial state* of the program that guarantees that the program state will satisfy the condition C every time control reaches the point p. We define the notion of *weakest at-least once pre-condition*, denoted as $wp_1(p, C)$, as the weakest constraint on the initial state of the program that guarantees that execution will reach p at least once in a state that satisfies C. Note that for any (p, C), $wp_1(p, C) = \neg wp(p, \neg C)$. We can use the weakest at-least once pre-condition to check if a selected dereference of a variable or access-path v at a given program point p is *always* safe. This can be done by checking if $wp_1(p, v = null)$ is *false*. However, the weakest at-least-once pre-condition is in general not computable precisely in the presence of loops or recursion; hence, our approach uses an abstract interpretation [1] to compute an over-approximation of it. After our analysis terminates, we check if the computed over-approximation of $wp_1(p, v = null)$ is *false*; if yes, it would imply that the precise solution is also *false*, implying the safety of the dereference. On the other hand, if the over-approximation is satisfiable, we declare the dereference as *potentially* unsafe.

Each element in the lattice that we use for abstract interpretation is a representation of a formula in disjunctive normal form, with each literal being a predicate that compares an access path with another access path or with *null*. An access path is a variable, or a variable followed by fields, e.g., $v.f_1.f_2...f_k$, that points to an object (i.e., is not of primitive type). The lattice elements are ordered by implication, where weaker formulas dominate stronger formulas; our join operation basically implements logical *or*. We illustrate our lattice as well as our analysis using an example, shown in Figure 1. Our notation is to show the formula that holds at the point above any statement to the right of the statement. Also, we enclose each *disjunct* in a formula (except the disjuncts *true* and *false*) within angle brackets, and indicate both conjunctions of predicates within a disjunct as well as disjunctions of disjuncts using commas. Note in the example that there are two disjuncts at the point above line 5, and a single disjunct at all other points. The underlining of certain predicates in the example can be ignored for now, and will be addressed later.

The input to our approach is a dereference that needs to be verified, which we refer to as the *root* dereference. In the example, the root dereference is that of `b.f` in line 9. Therefore, the first step in the approach is to initialize the formula at the point above line 9 to $\langle \texttt{b.f} = \mathit{null} \rangle$, as shown to the right of line 9. The analysis proceeds by propagating formulas in a backwards direction, using conservative transfer functions which over-approximate the weakest precondition semantics of each statement. The final result of this propagation at all points is shown in the figure. Assuming that the method `foo` is the entire program, the computed over-approximation of $\mathit{wp_1}(\mathit{line}\ 9, \langle \texttt{b.f} = \mathit{null} \rangle)$, which is shown adjacent to line 2, is *false*; hence, the root dereference is declared safe. We postpone a detailed discussion and illustration of our analysis to subsequent sections in the paper.

1.1. Challenges

The obvious advantage that a backwards analysis such as ours has over a forward counterpart is that it is *demand-driven*, meaning a single selected dereference can be verified. This is a very useful feature in a real world setting, where most changes are incremental and affect only a small part of a program. Thus, the developer will be able to verify the dereferences in the part of code that is modified, without paying the price of analyzing all dereferences in the program. There are several reasons why analyzing a single dereference in the backwards direction can be much more efficient than analyzing all the dereferences in the program using a forwards analysis; we postpone a detailed discussion of this to Section 2.5.

This said, a backwards analysis poses its own set of challenges. The first problem is that in order to obtain high precision we would need to perform *strong updates* on formulas that refer to fields of objects when they are propagated back through "put field" statements that write to fields of objects. However, techniques for performing strong updates on formulas have been proposed in the literature only for forward analyses. These techniques do not carry over naturally to the backward setting. The second problem is the resolution of *virtual calls* in large object-oriented programs. While a forward analysis could potentially use path-specific points-to information [2] to derive a precise set of targets for a virtual call, a backwards analysis would need to rely on imprecise *may points-to* information to identify an over-approximation of the candidate targets at a virtual call.

There are other problems we face that are shared by forward counterparts, too. Java programs make extensive use of libraries; entering and analyzing all library methods would take a heavy toll on the scalability of the technique. The usage of recursive data structures such as linked lists and trees, and the usage of arrays, pose challenges to any analysis, because code that uses these structures is hard to analyze precisely in an efficient manner. *Shape analysis* [3] is a sophisticated technique that has been proposed to handle recursive data structures, but it does not scale to programs of sizes we are interested in in its current state of evolution. Finally, context-sensitivity and path-sensitivity are typically required for precision, but can be complex or expensive to implement.

1.2. The base analysis

Our approach consists of two parts: 1) the base analysis, and 2) an extended analysis, which is the base analysis plus two major extensions. The base analysis was originally proposed, discussed, and evaluated by Madhavan and Komondoor [4]. The key elements of the base analysis are a base lattice of formulas, backwards transfer functions for each kind of statement that over-approximate the corresponding wp_1 semantics, as well as an inter-procedural component, based on a variant of Sharir-Pneuli's [5] tabulation based approach. We had earlier illustrated the base analysis (in the intra-procedural setting) using the example in Figure 1. The key conceptual novelty of the base analysis is a technique to perform strong updates by embedding aliasing hypotheses in the pre-conditions generated for statements, and using other preceding statements to validate or invalidate these hypotheses. For instance, in the example in Figure 1, the analysis produces two disjuncts at the point before Statement 5 from the post-condition after this statement, with each of the two disjuncts being the result of a strong update of the post-condition under a distinct aliasing hypothesis, namely, '$\texttt{b} \neq \texttt{c}$' or '$\texttt{b} = \texttt{c}$'. The first of these two disjuncts happens to later get invalidated at Statement 3. The base analysis also contains innovative techniques for context-sensitive and path-sensitive analysis which increase precision significantly.

1.3. Our first extension

The primary focus of this paper is the two extensions in the extended analysis, which address two of the challenges that the base analysis dealt with only in simple ways. The first challenge is virtual calls. The base analysis requires pre-computed may-points-to information to identify candidate targets of virtual calls. However, practical points-to analysis implementations typically compute imprecise information, which results in virtual calls having too many candidate targets, with many of them typically being spurious. A full analysis would therefore require a formula that reaches the point after a virtual call-site (i.e., a post-condition) to be propagated through all the candidate targets of the call. This would result in the analysis becoming very expensive. Yet, because of the spurious targets, the overall precision is lowered; for instance, a root dereference could be called unsafe even if it is actually safe if one of the spurious targets of a virtual call writes a *null* into the field referred to in the root dereference. The negative impact that virtual-call resolution has in general on scalability and precision of various kinds of analyses is well-known in the literature, e.g., as reported in these publications [6, 7, 2].

Therefore, to keep the analysis time practical, the base analysis is configured to *skip*, i.e., to not enter and analyze, any of the candidate target methods of virtual calls that have more number of candidate targets than a pre-set threshold. Instead, it identifies, using conservative *mod-ref* [8] information that can be efficiently pre-computed, a superset of the predicates in the post-condition whose truth value could be affected by the side-effects of the call or by the return value from the call. The analysis then computes the pre-condition (at the point before the call-site) as simply the set of unaffected predicates in the post-condition, based on the conservative assumption that the affected predicates could potentially all become true if the formula were to be propagated through the target methods. The pre-condition obtained in this way is clearly an over-approximation of the precise pre-condition. Therefore, eventually an over-approximation of the weakest at-least-once pre-condition is guaranteed.

However, an over-approximated pre-condition implies loss of precision. In the worst case, if all predicates get dropped, the analysis stops right away and declares the root dereference unsafe. Therefore the above mentioned technique of the base analysis, while allowing the base analysis to have a practical running time, results in significant loss of precision.

The first major extension that we employ is an alternative approach to the above technique that is typically more precise, while still expending less time than would be required to enter and analyze all the targets fully. Whenever we have a post-condition formula at the point after a difficult virtual call-site, we *carry* this formula directly to the statements that potentially write values into the field that is being compared with *null* in the formula, which we call the immediate producer statements, and then resume the base analysis from these statements using the carried formulas. For an illustration of this idea consider the formula to the right of line 9 in Figure 1. The immediate producers of this formula at this point are Statement 5, as well as any statements in the function that calls `foo` (not shown) that write a value into the field `b.f`. If line 9 were hypothetically preceded by a difficult virtual call-site, we would carry the formula $\langle \texttt{b.f} = \mathit{null} \rangle$ to its immediate producers. Considering the producer statement 5, the above formula after it is carried (and after a required rewriting) becomes the formula $\langle \texttt{t} = \mathit{null} \rangle$ at the point before this statement. The base analysis resumes from this point, resulting subsequently in this formula getting invalidated at Statement 4 itself.

Note that determining the immediate producers of a formula requires alias analysis. For instance, it is the aliasing introduced in Statement 3 in Figure 1 that causes Statement 5 to be an immediate producer of `b.f` at line 9. In our approach we pre-compute immediate producers of all access paths at all program points upfront, and use this information during the actual backwards null-dereference analysis.

Our technique of using immediate producers is typically more efficient than entering all targets of the virtual call, for the following reason. For the immediate producers that are in the calling method or its callees, as opposed to being in the target methods of the virtual call site, the analysis after resuming from these statements would not enter into the target methods at all. Other immediate producers could be inside the target methods; yet, even in this situation, only the portion of a target method that "precedes" an immediate producer needs to be analyzed. On other hand, this technique is typically more expensive than the simple mod-ref information-based technique resorted to by the base analysis, because (a) as mentioned above, certain portions of the target method may still need to be analyzed, which means the method cannot always be skipped entirely, and (b) in situations where the base analysis would have stopped because all the predicates in the disjunct were affected as per the mod-ref information, and would have called the root dereference unsafe, the extended analysis could still continue and eventually verify the root dereference as safe. Yet, the technique still has a practical running time requirement, while yielding significantly more precise results than the mod-ref-based technique.

Our technique is a fairly general one, in that it could potentially be used to side-step the analysis of any kind of difficult construct. In fact, we show its applicability in skipping not only difficult virtual calls but also library method calls, which the base analysis normally skips using mod-ref information because of the typical high complexity of analyzing library method implementations. The novelty of our technique is as follows. While certain previous approaches [9, 10, 11] have used the *transitive closure* of the immediate-producer relation (which is known as a *thin slice*) to perform certain inexpensive, approximate analyses, ours is the first to our knowledge to use immediate producers within the overall context of a precise, path-sensitive analysis to skip localized regions of code conservatively.

1.4. Our second extension

Our second extension also addresses the challenge of library methods, but in a different way. Here we target a *specific* category of frequently used libraries – the Java Collections API. Our approach is to use *summary functions* that we have designed manually as transfer functions at call sites to collections methods, rather than enter and analyze these methods or use conservative mod-ref information. The novelty of our approach, compared to previous related approaches [12, 13, 14, 15], is that our summary functions work in the backward direction, and are hence very different from the forward transfer functions used in these other approaches.

The impact of our two extensions is significant. Across 10 real-world medium to large sized Java programs on which the base analysis was originally evaluated, we find that the average percentage of dereferences reported as unsafe per benchmark has gone down from 16.25% by the base analysis to 9% by the extended analysis, which is a 45% reduction. Our extensions have caused the average running time to verify a dereference to go up from 288 ms to 427 ms; this happens, as discussed earlier, because our immediate producers extension is less efficient than the mod-ref technique of the base analysis. However, this running time requirement is still very reasonable, even in the context of an on-demand verification tool.

The rest of this paper is structured as follows. We give an overview of the base analysis in Section 2. In Section 3 we present our first extension, which is based on using immediate producers to skip portions of "difficult" programming constructs. We then present our second extension, to handle the built-in Java Collections APIs using manually provided summary functions, in Section 4.

Section 5 describes our implementation of our approach, while Section 6 has a discussion on the results of applying our implementation on real programs. Section 7 contains a discussion of related work, while Section 8 concludes the paper. An appendix follows, with two parts. Appendix A contains a proof of correctness of the base analysis, while Appendix B contains proofs of correctness of some of our key Collections API summary functions.

2. Base analysis

The base analysis checks if a given dereference is unsafe by computing an over-approximation of the weakest at-least-once precondition. In this section we give an overview of the abstract lattice and transfer functions used by this analysis, as well as a few other key features of this analysis. A detailed discussion of this analysis can be found in a previous publication [4].

2.1. Abstract Lattice and Transfer function

$$
\begin{array}{lcl}
\textit{Formula} & \equiv & 2^{\textit{Disjunct}} \\
\textit{Disjunct} & \equiv & 2^{\textit{Predicate}} \\
\textit{op} & \rightarrow & = \mid \neq \\
\textit{Predicate} & \rightarrow & \textit{AP op Atom} \\
\textit{Atom} & \rightarrow & \textit{AP} \mid \textit{null} \\
\textit{Fields} & \rightarrow & \textit{field.Fields} \mid \epsilon \\
\textit{AccessPath (AP)} & \rightarrow & \textit{Variable.Fields} \mid \textit{Variable}
\end{array}
$$

Figure 2: Structure of formulas (lattice elements) in base analysis

The data-flow lattice of the base analysis is described in Figure 2. Each lattice element is a *Formula*, which is a set of *Disjuncts*. Each *Disjunct* is a set of *Predicates*. Any $f \in \textit{Formula}$ is a set that can be logically interpreted as the disjunction of its elements; whereas, any $d \in \textit{Disjunct}$ can be interpreted as conjunction of its elements. Operands in *Predicates* are either *AccessPaths* (which point to objects), or *null*. Each *AccessPath* is either a *Variable* or a *Variable* followed by a sequence of fields. The ordering operation $\sqsubseteq$ of the lattice is defined as follows: $f_1 \sqsubseteq f_2$ iff $f_1 \subseteq f_2$, where $f_1, f_2 \in 2^{\textit{Disjunct}}$. Therefore, the join operator is set-union (which basically implements logical OR). The bottom element is the empty set of disjuncts (which represents logical falsehood), and the top element is the set of all Disjuncts (which represents logical truth). The lattice is made effectively finite by bounding the lengths of access paths. Whenever a predicate in the formula contains an access path in which some field repeats more than once in the sequence of fields, this predicate is *dropped* from the disjunct to which it belongs, i.e., is implicitly reduced to *true*. This in general results in an over-approximation, as the resulting formula after dropping a predicate is weaker than the original formula.

The program is assumed to be in an Intermediate Representation (IR) form like three-address code. The (backward) transfer functions for the individual

Name	**Instruction**	**Transfer Function**: $\lambda\phi \in Disjunct.\phi'$, where $\phi' \in 2^{Disjunct}$, and is =
COPY	$v = w$	$\phi[w/v]$
NULLASGN	$v = null$	$\phi[null/v]$
NEWASGN	$v = new\ T$	$\phi[t_i/v]$, where t_i is a variable representing all objects allocated at this instruction i
GETFIELD	$v = r.f$	$\phi[r.f/v] + \{r \neq null\}$
ASSUME	$assume(b)$	$\phi + \{b\}$, if b is "$AP\ op\ null''$ ϕ, otherwise
EXPRASGN	$v = v_1\ op\ v_2$	ϕ - $\{pred \in \phi \mid v \in Vars(pred)\}$
GETARRAY	$v = a[i]$	ϕ - $\{pred \in \phi \mid v \in Vars(pred)\}$
PUTARRAY	$a[i] = v$	ϕ
RETURN	$return\ v$	$\phi[v/ret]$, where ret is a place-holder for the return value
PUTFIELD	$r.f = v$	T,

where pseudo-code for computing T is as follows.

```
1: $T = \{\phi\}$
2: Let $SubAPs(\phi)$ be the set consisting of all prefixes (proper as well as
   improper) of all access paths that are operands of the predicates in
   $\phi$.
3: **for all** access paths $ap_i$ such that $ap_i.f \in SubAPs(\phi)$ **do**
4:     $S = T$
5:     **if** $MustAlias(ap_i, r)$ after "$r.f = v$" **then**
6:         **for all** $\phi_1 \in S$ **do**
7:             $T = T - \{\phi_1\} \cup \{\phi_2\}$,
8:               where $\phi_2 \equiv \phi_1[v/ap_i.f] + \{r \neq null\}$.
9:     **else if** $MayAlias(ap_i, r)$ after "$r.f = v$" **then**
10:        **for all** $\phi_1 \in S$ **do**
11:            $T = T - \{\phi_1\} \cup \{\phi_2\} \cup \{\phi_3\}$,
12:              where $\phi_2 \equiv \phi_1[v/ap_i.f] + \{r = ap_i\} + \{r \neq null\}$, and
13:                    $\phi_3 \equiv \phi_1 + \{r \neq ap_i\} + \{r \neq null\}$.
14:    **else** {$r$ and $ap_i$ are definitely not aliased}
15:        **for all** $\phi_1 \in S$ **do**
16:            $T = T - \{\phi_1\} \cup \{\phi_2\}$,
17:              where $\phi_2 \equiv \phi_1 + \{r \neq ap_i\} + \{r \neq null\}$.
```

Figure 3: Abstract transfer functions used in the base analysis

statements in the IR are shown in Figure 3. These functions are *distributive*; therefore, each function is expressed as taking a single disjunct ϕ in the statement's post-state as input, and returning a set (i.e., disjunction) of disjuncts ϕ' in the statement's pre-state. In all transfer functions other than the one for PUTFIELD instructions ϕ' contains a *single* disjunct; therefore, we omit the curly braces around this disjunct for convenience. We use the notation $\phi[w/v]$ to denote a disjunct that is identical to ϕ except that all instances of v have been replaced by w.

Every disjunct in the analysis has zero or one *root predicates.* The root predicate is always of form $AP = null$. At the start of the analysis, at the point just above the given root dereference the root predicate is the one that compares the access-path that is being dereferenced to *null*. Whenever a disjunct is propagated through an instruction the same predicate remains the root predicate, except that the access path in the predicate may get rewritten. For instance, this happens in line 5 in Figure 1, where `b.f` gets rewritten to `t` in one of the disjuncts. Our convention is to always underline the root predicate. Intuitively, the root predicate is an important predicate because it encodes the nullness hypothesis of the root dereference. The root predicate is therefore handled specially at various points in our analysis, as will become clear in the rest of this paper.

We now discuss the transfer functions shown in Figure 3. The functions Copy, NullAsgn, and Return are self-explanatory; we discuss below some of the more interesting ones. The ExprAsgn function *drops* all predicates in ϕ that depend on v ($\mathit{Vars}(\mathit{pred})$ denotes the set of program variables that occur in the predicate *pred*). This is because the base analysis abstracts away all the arithmetic from the disjuncts. The GetArray function does a similar reduction, because the base analysis does not model array accesses. Since ϕ can contain no array references, the PutArray function is basically an identity transfer function. NewAsgn uses the (standard) approach of representing all objects allocated at an allocation-site i by a single variable t_i (that is not present in the original program). We assume that "if" conditions in the source program are encoded in the IR using *assume* instructions in the standard way; i.e., the target of the *true* branch out of a condition "if (b)" is the instruction "*assume*(b)", while the target of the *false* branch is the instruction "*assume*$(\neg b)$". The base analysis implements a limited notion of path sensitivity, as follows: the Assume transfer function for the instruction *assume(b)* adds b to the disjunct if b is a predicate that compares an access path to *null*, and otherwise acts as a identity function.

2.1.1. Handling put-fields

As discussed in the introduction, the base analysis always performs *strong updates* at put-field statements for precision. At the instruction $r.f = v$, if ϕ is the postcondition, for each access path $ap_i.f$ in ϕ, the transfer function produces a separate disjunct in the pre-condition in the place of ϕ for each of the following two hypotheses: (i) r and ap_i refer to the same object, and (ii) r and ap_i do not refer to the same object. Under the first hypothesis a disjunct is generated by replacing all the instances of $ap_i.f$ in ϕ by v, and adding an *alias predicate* $\langle r = ap_i \rangle$ to it (see Lines 8 and 12 in the pseudo-code for the PutField instruction in Figure 3). Under the second hypothesis the analysis produces a disjunct that is the same as ϕ, with the *alias predicate* $\langle r \neq ap_i \rangle$ (see Line 13). Note that in order to distinguish them better we use the '+' symbol to add predicates to disjuncts (which are sets of predicates), and the '$\cup$' to union together sets of disjuncts. The alias predicate added to a disjunct embeds the aliasing hypotheses made in that disjunct, and can get validated or invalidated later in the analysis depending on the statements encountered in the path. This approach is entirely different from that taken in a typical forward analysis, where whether r and ap_i alias or not would be known by the time the analysis reaches the put-field statement along a path (or set of paths); therefore, different aliasing hypotheses need not be made.

For an illustration of the PutField function, see line 5 in the example in

$$\begin{array}{lll}
(1) & (ap = ap) & \longrightarrow \; \mathit{true} \\
(2) & \{ap_1 = ap_2, ap_1 \neq ap_2\} & \longrightarrow \; \{\mathit{false}\} \\
(3) & \{ap_1 = \mathit{null}, ap_1 \neq \mathit{null}\} & \longrightarrow \; \{\mathit{false}\} \\
(4) & (t_i = t_j) & \longrightarrow \; \mathit{false} \\
(5) & (t_i \neq t_j) & \longrightarrow \; \mathit{true} \\
(6) & (t_i = \mathit{null}) & \longrightarrow \; \mathit{false} \\
(7) & (t_i \neq \mathit{null}) & \longrightarrow \; \mathit{true} \\
(8) & (t_i = ap) & \longrightarrow \; \mathit{false} \\
(9) & (t_i \neq ap) & \longrightarrow \; \mathit{true}
\end{array}$$

Figure 4: Rules for simplifying disjuncts

Figure 1. Note that the single disjunct at the point after this line gets turned into two disjuncts at the point before this line.

Note that the PUTFIELD transfer function makes use of a utility function $\mathit{SubAPs}(\phi)$; this returns a set consisting of all prefixes (proper as well as improper) of all access paths that are operands of the predicates in ϕ. For instance, $\mathit{SubAPs}(\langle v.f = \mathit{null}, v = u.g\rangle)$ is $\{v, v.f, u, u.g\}$. Note also that the transfer function makes use of pre-computed *MayAlias* and *MustAlias* information (if available). Using must-aliasing information, if available, is an optimization for efficiency with no influence on the precision of the base analysis. If must-alias information is not available then the analysis assumes that no two access paths at a program point are must-aliased unless they are syntactically identical.

2.1.2. Simplification rules

Inspired by the Snugglebug [2] approach, rather than use a theorem prover, the base analysis uses a lightweight custom simplifier on each disjunct after it is produced by a propagation step to validate, invalidate, or simplify the disjunct. Figure 4 shows a sampling of the rules used in the simplifier. Rules 2 and 3 reduce an entire disjunct to *true*/*false*; the other rules reduce individual predicates (in disjuncts) to *true*/*false*. For instance, the disjunct just above statement 2 in Figure 1 is reduced to *false* by applying Rule 2. In the rules t_i is a special variable that represents objects allocated at a static allocation site i. The simplification rules are conservative; i.e., they may simplify a disjunct to something weaker than what is ideally possible. Thus, the soundness of the analysis is preserved.

2.2. Example

We use the example in Figure 1 to illustrate the base analysis. The root dereference is the dereference of `b.f` at line 9; hence we start the analysis with the singleton disjunct $\langle$`b.f` $= \mathit{null}\rangle$ at the point above this dereference, and the empty set of disjuncts at all other points. Predicate `d` $\neq \mathit{null}$ gets added to the pre-condition above line 8 (for path-sensitivity) by the transfer function ASSUME. `d` gets rewritten to `a` in line 7. Then, the disjunct `b.f` $= \mathit{null}$, `a` $\neq \mathit{null}$ at point above line 7, while propagating through the *true* branch of the conditional at line 2, encounters a putfield statement in line 5, hence resulting in two disjuncts above line 5. The second of these two disjuncts gets invalidated at line 4, while the first one gets invalidated at line 3. Thus, *false* (which we use to denote the empty disjunct) reaches the point above line 3. The disjunct above line 7 also propagates through the *false* branch of the conditional at line 2, the result of which is shown to the right of line 2. This disjunct gets simplified to *false*. Therefore, since only *false* reaches the point above line 2 along both branches, the dereference in line 9 is reported as safe; in this example, this turns out to be the precise weakest at-least-once pre-condition.

2.3. Interprocedural analysis

The inter-procedural aspect of the base analysis is modeled on that of the *Xylem* [16] approach, which itself is based on Sharir and Pneuli's tabulation based approach [5]. The analysis follows a *depth-first* propagation. That is, when a disjunct ϕ reaches the point that follows a call-site to a method m in a method n, the analysis of method n (i.e., the propagation of disjuncts within method n) is suspended. The disjunct ϕ is transformed by replacing each occurrence of an actual parameter at the call-site with the corresponding formal parameter of m, and then propagated to the end of m's body. This disjunct is propagated up through m (and in a similar way, through its transitive callees). The set of disjuncts that thus results at the entry of m is finally propagated back to the point that precedes the call-site to m in n (with a corresponding inverse replacement of formal parameters with actual parameters), and the analysis of method n is resumed. When a virtual callsite is encountered, pre-computed may-points-to information is used to resolve the potential targets of the call.

For the sake of efficiency, as well as to ensure termination, the analysis maintains a summary table $\Sigma[\![m]\!] : \mathit{Disjunct} \rightarrow 2^{\mathit{Disjunct}}$, which is a partial map, which associates each disjunct ϕ that was propagated to the exit of method m with the set of disjuncts that would result at the entry of m upon propagating ϕ through the method m (and its transitive callees). The summary table is used to avoid re-analysis of methods that are entered multiple times with the same post-condition.

The approach discussed above is outlined as pseudo-code in Figure 5. The routine *wpWrtMethod* shown here computes the pre-condition at the entry of a given method m given a post-condition ϕ_{post} at the exit of m. We will discuss the need for the context-stack *CS* later.

A related point is that when the root dereference is in a method m, or in a callee of a method m, and a propagated disjunct reaches the entry of m, then there is no specific caller of m to return to. Therefore, the disjunct is propagated to the point before *each* call-site in the program that calls m (after a transformation of formal parameters to actual parameters, as described earlier). We call the methods that contain these call-sites the *predecessors* of m. Similar to the summary table Σ, the analysis also maintains a set *propagated* to keep track of disjuncts propagated from each method m to its predecessors during analysis of root dereferences in m (or its callees), and avoids repeated propagation of such disjuncts.

```
1: **Procedure** $wpWrtMethod(m, \phi_{post})$
2: $ex$ = exit statement of $m$
3: worklist $W = \{(ex, \phi_{post})\}$
4: **if** $\Sigma[\![m]\!](\phi_{post})$ is not defined **then**
5:     update $\Sigma[\![m]\![\phi_{post} \mapsto$ emptyset$]$
6: push($CS$,$(m, \phi_{post})$)
7: $\Gamma_m = \Sigma$ {$\Gamma_m$ is a snapshot of summary table $\Sigma$ before analysis of $m$ and its
   callees.}
8: **while** $W \neq \emptyset$ **do**
9:     Select $(S, \phi) \in W$ {$\phi$ *is a post-condition after stmt* $S$}
10:    **if** $S$ is a call instruction $v_r = c(v_1, v_2, \ldots, v_n)$ **then**
11:        **if** $(c, \phi) \in CS$ **then**
12:            $output = \Sigma[\![c]\!](\phi)$
13:        **else**
14:            **if** $\Sigma[\![c]\!](\phi)$ is defined **then**
15:                $output = \Sigma[\![c]\!](\phi)$ {*Summary hit*}
16:            **else** {*Summary miss*}
17:                $output = wpWrtMethod(c, \phi)$
18:    **else**
19:        Propagate $\phi$ back through $S$. Let $output$ be the set of disjuncts obtained
           (at the point preceding $S$) as a result.
20:    For each disjunct $\phi'$ in $output$ and for each predecessor statement $S'$ of $S$
       add $(S', \phi')$ to $W$.
21: Let $result$ be the set of disjuncts that results at the entry of method $m$ due
    to the propagation above.
22: $pop(CS)$
23: **if** $(\Sigma[\![m]\!](\phi_{post}) \neq result)$ **then**
24:     $\Sigma = \Gamma_m$ {replace the summary table by the saved snapshot}
25:     update $\Sigma[\![m]\![\phi_{post} \mapsto result]$
26:     $result = wpWrtMethod(m, \phi_{post})$
27: **else**
28:     update $\Sigma[\![m]\![\phi_{post} \mapsto result]$
29: return $result$
```

Figure 5: Computing $wp_1(ex, \phi_{post})$ at the entry of method m for post-condition ϕ_{post} at the exit statement ex of m.

The above-described depth-first approach (which is inherited from Xylem) gives two important benefits over a breadth-first or chaotic iteration approach: (a) It prioritizes exploration of long paths, which are often required to be traversed in order for certain safe root dereferences to be identified as such. (b) It minimizes usage of space, by requiring control-flow graphs (CFGs) and other analysis artifacts of only those methods that are in a single calling-sequence to be kept in memory together at a time.

2.3.1. Handling recursion

Consider once again the routine *wpWrtMethod* in Figure 5, to be run on a method m with a given post-condition ϕ_{post}. Since m or its callees could be recursive or mutually recursive, and since analysis of m triggers analysis of m's callees using recursive invocations of *wpWrtMethod* (see line 17 in the pseudo-code), *wpWrtMethod* could go into non-termination whenever a method c (c being m or a callee of m) needs to be analyzed wrt a post-condition ϕ during the context of an unfinished analysis of c wrt the same post-condition.

Therefore, in order to ensure termination, the first thing the routine *wpWrt-Method* does is to push its arguments (m, ϕ_{post}) into a context-stack *CS* (see line 6 in the pseudo-code). Later, whenever the pre-condition of a method c is required to be computed wrt to post-condition ϕ, and (c, ϕ) is already in the context-stack (as checked for in line 11 in Figure 5), the analysis picks up the (potentially intermediate non-fix-point) result $\Sigma[\![c]\!][\phi]$ from the summary table (instead of starting a re-analysis of c), and continues with the analysis of the caller.

Eventually, when the original invocation of *wpWrtMethod* on (m, ϕ_{post}) terminates, the following steps are taken (see line 21 onward in the pseudo-code): (1) $\Sigma[\![m]\!][\phi_{post}]$ is set to the disjuncts that were propagated to the entry of m. (2) All other updates made to the summary table Σ by this original invocation or by its recursive invocations are *undone* (because they are potentially derived from intermediate non-fix-point entries in the summary table). (3) If the newly updated value in $\Sigma[\![m]\!][\phi_{post}]$ is not equal to the corresponding pre-existing value then an analysis of method m with post-condition ϕ_{post} is *again* started. Note that in the re-started analysis mentioned above, if a call-site to m with post-condition ϕ_{post} is again encountered, then the updated value in $\Sigma[\![m]\!][\phi_{post}]$ is used. This process is repeated iteratively until the value in $\Sigma[\![m]\!][\phi_{post}]$ stabilizes. In the final iteration of this activity steps (2) and (3) mentioned above are not performed.

2.4. Some details about library calls

Library methods are typically large and complex, and hence are expensive to analyze. Moreover, they often use recursive data structures, which cause formulas to be generated with repeating fields in access paths, as well as arrays; as discussed earlier in this section, these constructs are not modeled precisely enough by the base analysis.

Therefore, the base analysis uses a few conservative heuristics to minimize analysis of library methods. First of all, a library method is entered and analyzed in the normal manner from the point that follows a call-site to the method *only if* the variable that gets assigned the return value from the call is involved in the post-condition at this point. Else, the target library method(s) at the call-site are skipped, as follows. First, pre-computed mod-ref information [8] is used to identify the actual parameters of the call from which it is potentially possible to reach objects that could potentially be modified by the target method(s). Next, the pre-condition at the point before the call-site is set to be equal to the set of predicates in the post-condition that do not involve any of the affected actual parameters identified above.

On a related note, the base analysis uses a manually constructed list of library methods, called a *skip list*, which lists library methods that are side-effect free as per their specification. For these methods the (often imprecise) pre-computed mod-ref information is ignored; i.e., the pre-condition is set to be the same as the post-condition (unless the return value from the method call is used in the post-condition, in which case the method is entered and analyzed). The skip list currently contains around 136 methods.

There is yet another manually constructed list, called the *analyze list*, which is non-overlapping with the skip list, and contains library methods that are considered to have important side effects. The methods in this list are *always* entered and analyzed, even if the return value from the method-call is not used in the post-condition. Currently the analyze-list contains around 84 library methods.

2.5. Advantages of a backwards analysis

Approaches for null-dereference verification that preceded our publication [4] on the base analysis, for instance, [17, 18], are based on a *forward* analysis, and are meant to verify *all* dereferences in a program in one go. The base analysis is *demand-driven* in many ways; i.e., it only does work that is required to verify the given root dereference. (a) It keeps track only of aliasing relationships that are pertinent at the put-field instructions traversed so far along the path being currently analyzed. A forward analysis would potentially need to eagerly track *all aliasing* relationships to achieve similar precision. (b) It keeps track of null-ness or non-null-ness information only for the access paths that occur in the root dereference or in "if" conditionals traversed so far. A forward analysis might need to track this information for all access paths eagerly. (c) It only analyses paths from the program entry to the root dereference; other paths are never traversed. (d) It can stop and declare the root dereference to be safe or unsafe much before propagation reaches the program entry. In particular, the base analysis declares a root dereference safe as soon as all extant disjuncts get invalidated at some point during the analysis. In fact, in large programs, more than 95% of dereferences that were reported as safe needed only paths of length up to 50 instructions to be traversed. Conversely, the analysis declares the root dereference unsafe as soon as a *true* disjunct reaches the entry point of any method (although, in general, there are situations wherein continuing the propagation of this disjunct to callers of the method could result in an eventual invalidation of this disjunct). It is this demand-driven nature of the base analysis that enables it to have an extremely low response time, of around 288 ms on average for the analysis of a single dereference. Our extended analysis, which we focus on in the remaining sections of this paper, entirely preserves this backward and demand-driven style of analysis, while incorporating extensions to enhance its precision.

3. Improving our analysis using producer-consumer edges

As has been discussed in the Sections 1.3 and 2.4 there are two *limits* that the base analysis employs to keep the running time within practical bounds. (a) When a virtual call-site has greater number of candidate targets than a pre-set threshold (which was 10 in our experiments), the pre-condition is computed simply by dropping from the post-condition the predicates that are potentially affected by the side-effects of any of the target methods of the call-site (as per pre-computed mod-ref information) or that refer to the return value. (b) When a library call is encountered, and the return value from the call is not referred to in the post-condition, then the call is similarly skipped using mod-ref information.

There is also a third limit (c) employed by the base analysis, as follows. A *library call-back* method (which we often abbreviate simply as a call-back method) is an application method that is called by library methods; e.g., `equals` methods in user-defined classes are called by library methods such as `HashSet.add`. If the given root dereference is inside a call-back method, then during the analysis a formula could get propagated to the entry point of the call-back method. Now, rather than propagate this formula via the library code back to the application code (i.e., to its calling contexts), the base analysis is configured to give up entirely and call the root dereference unsafe. (Note that the over-approximation strategy using mod-ref information is not applicable in this setting, because the formula to be propagated is not at a point that follows a call-site.)

These limits were found essential to keep the base analysis practical. On a set of eight real benchmarks it was observed that if the two limits mentioned above are not enforced then the analysis time went up on average by *87 times* per dereference. (We discuss in more detail our experimental setup as well as these results in Section 6.) Nonetheless, these limits are a significant contributor to the overall imprecision in the base analysis. With call-backs it is easy to see why the limit causes imprecision. The example in Figure 6 illustrates a simple scenario where the virtual-call limit causes imprecision. Say the virtual call in line 8 has more targets than the threshold, with the method `bar` in lines 10-14 being one of the targets. Say the root dereference is the dereference of `z.f` at Statement 9. Therefore, we have the formula $\langle \texttt{z.f} = \mathit{null} \rangle$ at the point p that precedes Statement 9. The base analysis would immediately reduce this formula to *true*, because it involves the return value `z` from the call, and hence end up calling the dereference unsafe. However, it is clear that the root dereference is actually safe (ignoring, for simplicity, the other targets of the virtual call). This is because the only values that flow to `z.f` at line 9 are the values that are put into `u` and `v` in Statements 2 and 3, respectively, which are both non-null.

```
1:  foo() {
      ...
2:    B u = new B();
3:    B v = new B();
4:    A x = new A();
5:    x.f = u;          ⟨u  =  null⟩
6:    A y = new A();
7:    y.f = v;          ⟨v  =  null⟩
8:    A z = w.bar(x, y);
9:    print z.f.g;      ⟨z.f  =  null⟩ // point p
    }

10:  bar(A x, A y) {
11:   A z = x;
12:   if (x == null)
13:     z = y;
14:   return z;
    }
```

Figure 6: Example to illustrate handling of difficult virtual calls. The root dereference is in bold.

Our approach, in brief, is to directly *carry* the predicate $\langle$`z.f` $= null\rangle$, which the base analysis would have otherwise reduced to *true* by employing its limit, from the point p to the points that precede the statements that write into the location referred to by `z.f` at point p. These statements happen to be the ones in lines 5 and 7; the carried predicates (after a required rewrite step) are shown to the right of these statements. The carried predicates are subsequently checked using (independent) applications of the base analysis. In this case they are both found to be infeasible, thus indicating that the original post-condition $\langle$`z.f` $= null\rangle$ at the point after the virtual call is infeasible. Therefore, the root dereference is declared safe.

On a set of ten benchmark programs (the eight programs mentioned earlier, plus two more), on average per program, at least one of the limits in the base analysis kicked in and dropped the root predicate during the analysis of about 26% of the dereferences in the program that were eventually declared unsafe. In particular, 13% experienced the virtual-call limit, 10% experienced the call-back limit, while 3% experienced both limits (along different paths). These 26% of unsafe dereferences are intuitively the ones that were called unsafe due to the limits. Our approach achieves significant improvement in precision over the base analysis. Our approach is able to declare 68% of *these* 26% of dereferences as safe. This result basically confirms our hypothesis that virtual calls and call backs were a significant cause of imprecision in the base analysis. At the same time, the increase in running time is very reasonable, as was mentioned in Section 1.3.

3.1. Immediate producers

Our extension is based on the notion of *immediate producers*, which is defined as follows: A statement q is an immediate producer of an access path AP at a program point p iff (1) there exists a memory location l that may be referred to by AP at point p, (2) the content of location l is *not* dereferenced by AP to obtain another address, (3) q is an assignment statement, and l is one of the addresses that q may write into, and (4) there is a path from q to p along which l is not written to by any statement. Intuitively, the immediate producers of an access path at a point are the statements that assign to the locations referred to by the final (innermost) field in the access path.

For instance, in the example program in Figure 6, Statements 5 and 7 (which are underlined) are the immediate producers of the access path `z.f` at the point before Statement 9. Note that Statements 11 and 13 are *def-use* [8] (or flow-dependence) predecessors of `z.f` at the point that precedes Statement 9, but are *not* immediate producers, because the value in `z` is used only for dereferencing in `z.f`. Note that if we disregard condition (2) in the definition above what remains is nothing but the definition of *def-use* predecessors. Therefore, the immediate producer relation is a restricted form of the def-use relation. Note that identifying the immediate-producers of an access path at a program point requires pre-computed aliasing information (as does identifying the def-use predecessors of an access path). In the example in Figure 6, Statements 5 and 7 cannot be identified as the immediate producers of the access path `z.f` at Statement 9 without taking into the account the aliasing relationships established in Statements 11 and 13.

Our notion of immediate producer is derived from the notion of *producers*, introduced by Sridharan et al. [19]. For them, a producer of a statement s is any statement q such that q copies a value that may eventually flow to s and be used at s for a purpose other than pointer dereferencing. Note that if we define the immediate producers of any statement s="*AP1 = AP2*" as the immediate producers of *AP2* at the point before s, then the producer relation (between statements) is nothing but the reflexive transitive closure of this immediate producer relation between statements. In the example in Figure 6, if Statement 9 had been simply "`print z.f`", then the underlined plus wavy-underlined statements would be its producers. The underlined statements would be the immediate producers, while the wavy-underlined statements would be the transitive producers.

Lemma 1. *A predicate "*AP $= k$*", where k is any constant value, can be true at a program point p* only if *at least one of the immediate producers of* AP *at p could write the value k.*

For instance, in Figure 6, the predicate shown to the right of Statement 9 can be true only if one of the predicates shown to the right of Statements 5 or 7 can be true at those respective points. The correctness of this observation is easy to see, because the value referred to by AP whenever execution reaches point p is guaranteed to be the value written by the most recent preceding instance of one of the immediate producers of AP at p. The extension that we discuss in this section is based on this observation.

Sridharan et al. [19] describe an approach to compute an over-approximation of the immediate producer relation between statements. In our approach we

```
Subroutine: carryDisjunctToImmProds
Input: (p,$\phi$), where p is a program point and $\phi$ is a disjunct at p.
Output: $\{(s_i, \phi_i) \mid s_i$ is an immediate producer of $AP$ at $p$, where "$AP = null$" is
the root predicate of $\phi$, and $\phi_i$ is this root predicate carried over to the point that
precedes $s_i$.$\}$
Step 1:  $result = \emptyset$
Step 2:  IF $\phi$ has a root predicate
Step 3:    $rootAp = getRootAP(\phi)$
Step 4:    $prodSet = ImmProds(rootAp,p)$
Step 5:    FOR ALL statement $s$ in $prodSet$
Step 6:      $ap = getRhsAP(s)$
Step 7:      $\phi' = \{\langle ap = null\rangle\}$
Step 8:      $result = result \cup \{(s,\phi')\}$
Step 9:    END FOR
Step 10: END IF
Step 11: RETURN $result$
```

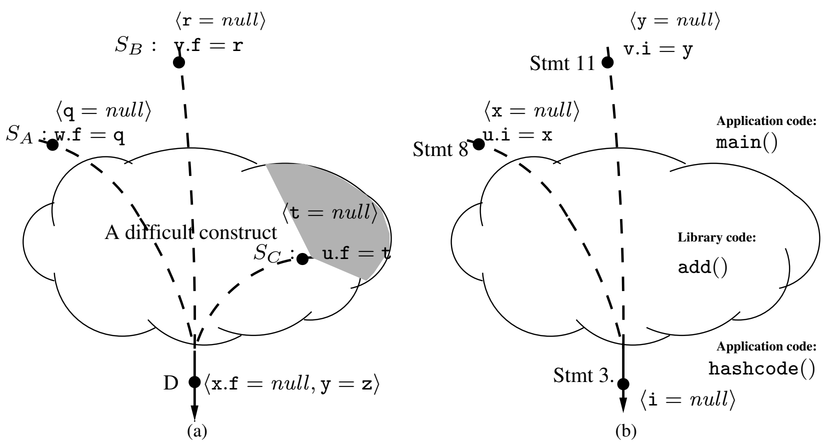
Figure 7: Subroutine *carryDisjunctToImmProds* carries disjunct ϕ at point p to the immediate producers of the access path in the root predicate in the disjunct.

assume that this relation is pre-computed for all access paths at all program points. While there is some cost associated with this, we have observed in practice that this is much less than the expense involved in analyzing all targets of a virtual call or in analyzing library methods from entry points of call-back methods. We provide some additional details about how we determine the immediate producers of statements in Sections 5.

3.2. Carrying disjuncts using immediate producers

Figure 7 shows a subroutine *carryDisjunctToImmProds* which we use in our technique. Its objective is to take a disjunct ϕ at a point p as input, and to "carry" ϕ to the points that precede the immediate producers of ϕ at p. Since Lemma 1 is applicable only to a single predicate, the subroutine first drops all other predicates from ϕ, and retains only the root predicate. This it does by invoking the utility method *getRootAP(ϕ)*, which returns the access path that occurs in root predicate in the disjunct ϕ (see Step 3). It then invokes the routine *ImmProds(rootAp,p)* (see Step 4), which implements Sridharan et al.'s algorithm [19] and returns the immediate producers of *rootAp* at point p. The algorithm then goes over all the immediate producers (see the loop in Steps 5-9). Corresponding to each immediate producer s, it constructs a single-predicate disjunct $\{\langle ap = null\rangle\}$ at the point before s, where ap is the rhs of the statement s. Note that as per Lemma 1 ϕ can be true at point p only if at least one of these created disjuncts can be true at its respective point. The algorithm returns a set of pairs, each one consisting of an immediate producer statement and the disjunct constructed at the point before this statement.

We illustrate Subroutine *carryDisjunctToImmProds* in Figure 8(a). Here, point D is the one that's preceded by a difficult construct (denoted by the cloud), with the disjunct at this point being $\langle \underline{\texttt{x.f} = null}, \texttt{y} = \texttt{z}\rangle$. "$\texttt{x.f} = null$"



$\langle$r = *null*$\rangle$
S_B : y.f = r
$\langle$q = *null*$\rangle$
S_A : w.f = q
A difficult construct
$\langle$t = *null*$\rangle$
S_C : u.f = t
D $\langle$x.f = *null*, y = z$\rangle$
(a)
$\langle$y = *null*$\rangle$
Stmt 11 v.i = y
$\langle$x = *null*$\rangle$
Stmt 8 u.i = x
Application code:
main()
Library code:
add()
Application code:
hashcode()
Stmt 3.
$\langle$i = *null*$\rangle$
(b)

Figure 8: (a) Propagation of the disjunct at D to sites producing values for it on encountering *Difficult construct*. (b) Illustration of our approach on the example in Figure 10.

being the root predicate, we first find the immediate producers of x.f at point D. Say these are the statements s_A, s_B, and s_C. We drop the predicate "y = z", and carry the root predicate from D to the points above these three statements. The thus carried predicates are shown in the figure within angle brackets at the points that precede the three immediate producer statements, respectively. In this case, s_A and s_B are outside the difficult construct, while s_C is inside. Note that the reason we drop the predicate "y = z" is that, in general, we do not know the variables at the points that precede s_A, s_B, and s_C, respectively, that store the values that eventually flow into y and z at point D. Note also that in each carried predicate the root predicate now involves the rhs of the corresponding immediate producer statement.

Algorithm *IsDisjunctInvalid* in Figure 9 is the main routine of our extension. It also takes a disjunct ϕ at a program point p as input. It is invoked from the base analysis whenever a disjunct ϕ reaches a point p that is just after (any kind of) difficult construct. (How our approach specifically identifies difficult constructs at which to apply this routine is the topic of Section 3.3). This routine first uses *carryDisjunctToImmProds* to carry ϕ to points that precede the immediate producers of ϕ at p (see Step 1). It then applies an (independent) instance of our complete backwards analysis on each of the carried disjuncts (Step 7), and returns *true* iff each of these disjuncts got invalidated. Upon its return the base analysis invalidates ϕ if the routine returned *true*, else it proceeds to propagate ϕ from p in the normal manner (i.e., ignoring our extension).

Note that in Steps 2 and 3 the algorithm returns *false* if there are no immediate producers; this is actually to address an implementation-specific corner case, which we discuss in 5.2.

Reverting to the illustration in Figure 8(a), this algorithm causes the disjunct at point D to be carried to points s_A, s_B, and s_C, as shown. It causes

```
Algorithm: IsDisjunctInvalid
Input: ($p$,$\phi$), where $p$ is a program point and $\phi$ is a disjunct at $p$.
Output: true, if $\phi$ can be invalidated, false otherwise.
Step 1:  carriedDisjSet = carryDisjunctToImmProds($p$,$\phi$)
Step 2:  IF carriedDisjSet is empty
Step 3:    result = false
Step 4:  ELSE
Step 5:    result = true
Step 6:    FOR ALL ($s$,$\phi'$) $\in$ carriedDisjSet
Step 7:      Apply an instance of our complete analysis
             treating $\phi'$ at the point before $s$ as the root post-condition.
Step 8:      IF the above analysis did not invalidate $\phi'$
Step 9:        result = false
Step 10:         BREAK
Step 11:       END IF
Step 12:    END FOR
Step 13: END IF
Step 14: RETURN result
```

Figure 9: Algorithm *IsDisjunctInvalid* checks if the disjunct ϕ at point p can be carried to its immediate producers and then invalidated.

each of these disjuncts to be treated as a post-condition and to be analyzed using our complete analysis. The shaded area inside the cloud represents the portion of the difficult construct that needs to be analyzed fully. For a concrete illustration consider the example in Figure 6. The disjunct at the point after Statement 8 is first carried to the points before Statements 5 and 7, as shown to the right of these statements. These two disjuncts are then analyzed separately, during which process they get invalidated (at Statements 2 and 3, respectively). Now, consider a variant of this example such that Statement 5 is inside method `bar`, between Statements 11 and 12. In this scenario the carried disjunct $\langle$`u` = *null*$\rangle$ would have been propagated through Statement 11, and then via Statements 7, 6, etc., until it reaches Statement 2 and gets invalidated.

The following theorem states the soundness of our extension.

Theorem 1. *Algorithm IsDisjunctInvalid(p, ϕ) returns true only if $wp_1(p, \phi)$ is false.*

The correctness of this theorem is easy to argue. Let *rootAp* be the root predicate of ϕ. (a) Assuming that subroutine *ImmProds(rootAp, p)* returns an over-approximation of the true set of immediate producers of *rootAp* at p (which is guaranteed by the approach of Sridharan et al. [19]), and assuming that the rest of our analysis is correct (i.e., over-approximates the weakest at-least-once precondition), it can be easily seen that Algorithm *IsDisjunctInvalid* returns *true* only if the weakest at-least-once pre-conditions of all disjuncts that are obtained by carrying *rootAp* to its immediate producers are *false*. (b) In other words, the algorithm returns *true* only if none of the immediate producers of *rootAp* at point p copy a *null*. (c) Now, from Lemma 1, it follows that the

```
1: public class Element {
2:   Integer i;
3:   public int hashCode() {         Entry of Call-Back hit !!!
4:    return i.hashCode();            ⟨i = null⟩
     }
   }

   main() {
5:   z= new HashSet();
6:   Element u = new Element();
7:   Integer x = new Integer(0);   false
8:   u.i = x;                      ⟨x = null⟩  (immediate producer of i at line 4)
9:   Element v = new Element();
10:  Integer y = new Integer(1);   false
11:  v.i = y;                      ⟨y = null⟩  (immediate producer of i at line 4)
12:  z.add(u);
13:  z.add(v);
   }
```

Figure 10: Example to illustrate verification of a root dereference (indicated in bold) inside a call-back method.

algorithm returns *true* only if *rootAp* cannot be null at p. This implies that $wp_1(p, \phi)$ is *false*. □

3.3. Instantiation of our approach

In our current implementation we treat virtual calls with too many candidate targets, certain library calls, and entry points of call-back methods as points at which to employ our extension by invoking the routine *IsDisjunctInvalid*. We state this idea more precisely below.

Whenever the base analysis chooses to skip a target of a call-site and use the target's mod-ref information instead, either because the target was a library method whose return value is not used in the post-condition or because the call-site had too many candidate targets, and pre-computed mod-ref information of this target reveals that it may affect the root-predicate of the disjunct ϕ at the program point p that follows the call-site, then the extended analysis over-rides the base analysis at this call-site. It does not enter and analyze *any* of the targets of the call-site; instead it calls the routine *IsDisjunctInvalid*(p, ϕ), and proceeds when this routine returns as was discussed in Section 3.2. A similar action is performed whenever a disjunct ϕ reaches the entry point p of any library call-back method.

We have already illustrated the handling of virtual calls by the extended analysis using the example in Figure 6. To illustrate the handling of call-backs, consider the example in Figure 10. Here, the method `Element.hashCode` (defined at lines 3–4) is a library call-back, as it is invoked by the library method `HashSet.add`, which itself is invoked on the variable `z` in Statements 12 and 13. In this example, say our root dereference is that of field '`i`' in Statement 4 in the method `Element.hashCode`. The analysis begins by propagating the disjunct

$\langle i = null \rangle$ to the entry of this method. At this point, rather than simply reduce this disjunct to *true* (hence calling the root dereference unsafe), it gets carried to the points that precede its immediate producers, which are Statements 8 and 11. The result of this carrying is shown in Figure 10, as well as in Figure 8(b). (In fact, in this example, we have obtained the exact same precise outcome without analyzing any library code as would have been obtained if the base analysis had hypothetically been able to propagate the disjunct at the entry of the method `Element.hashCode` via the code of the library method `HashSet.add` with full precision to the points before the calls to `add` in the `main` function.) The two carried disjuncts both get subsequently invalidated by subsequent applications of our complete analysis. Therefore, the dereference at Statement 4 is called safe.

It is somewhat difficult for us to measure, with virtual calls, the percentage of times that all immediate producers of a post-condition after the call-site are outside all target methods of the call-site as well as their transitive callees. On the other hand, we are able to measure this for call-backs. Our experiments using the ten real benchmark programs that we referred to earlier reveal that 95% of the time when a disjunct reaches the entry of a library call-back method it refers to locations that are written to only in application code (and thus ends up being carried directly to application code).

3.4. Precision of our approach

Our experiments on real benchmarks, which we discuss in detail in Section 6, show clearly that in practice our extended analysis is more precise than the base analysis. In fact, it is more precise in practice than even the base analysis without any limits (even though such an analysis is impractically expensive). However, these gains are not guaranteed to hold in every single program. We now present a series of examples to illustrate various scenarios. The example in Figure 6 illustrated how the extended analysis gives more precision than the base analysis with limits. In this same example, if the limits in the base analysis were removed, and if the other targets of the call in Statement 8 (not shown in the figure) all are such that they can be analyzed precisely by the base analysis, then the base analysis would also prove the root dereference safe, as did the extended analysis.

To see that the extended analysis analysis can outperform the base analysis even without any of its limits, consider the variant of the example in Figure 6 that is shown in Figure 11(a). Consider the performance of the base analysis without any limits. The predicate shown to the right of Statement 12 gets propagated to the point before Statement 20 when the analysis enters method `bar`. Later, when this predicate reaches the point after Statement 17, then the GetArray transfer function of the base analysis (see Figure 3) drops it (because the contents of arrays are not modeled precisely). Hence, the root dereference is called unsafe. However, the extended analysis directly carries the predicate at the point after Statement 11 to the points that precede Statements 7 and 9, as shown to the right of these statements in the figure. Intuitively, even though arrays are still not being modeled precisely, the analysis is able to

```
1:  foo() {
      ...
2:    B u = new B();
3:    B v = new B();
4:    for(i = 0; i < n ++i) {
5:      a[i] = new A();
6:      if (..)
7:        a[i].f = u;  ⟨u  =  null⟩
8:      else
9:        a[i].f = v;  ⟨v  =  null⟩
10:   }
11:   A z = w.bar(a);
12:   print z.f.g;     ⟨z.f  =  null⟩
    }

13: bar(A[] a) {
     ...
14:   A z = a[0];
15:   while (i < n) {
16:     if (..)
17:       z = a[i];
18:     i++;
19:   }
20:   return z;
    }
```

(a)

```
1:  foo() {
     ...
2:   a.f.g = null;
3:   if (y != null)
4:     a.f.g = new B();
5:   z = a;             {⟨a.f.g  =  null⟩,
                          ⟨y  ≠  null⟩}
6:   if (y != null) {
7:     copies z.f to itself
8:     print z.f.g.h;        ⟨z.f.g  =  null⟩
     }
   }
```

(b)

Figure 11: (a) Example to illustrate imprecision due to arrays. (b) Imprecision due to immediate producers.

determine that the `f` field of *each* element of the array `a` points to either what variable `u` did or what variable `v` did. From this point on the base analysis takes over, and is able to invalidate the predicates shown to the right of Statements 7 and 9. Therefore, the root dereference is declared safe.

We hypothesize that the arrays and recursive data structures are the chief reasons why the base analysis shows a large increase in running time but little improvement in precision when its limits are removed. On the other hand, the extended analysis is often able to side-step usages of these constructs in a way that retains precision, as is evidenced by our empirical results.

In the converse direction, it is possible to come up with examples in which the base analysis without limits performs more precisely than the extended analysis. Consider the example in Figure 11(b). The initial formula due to the root dereference is shown to the right of Statement 8. Let line 7 denote a call to *some* method that ends up copying the value in the field `z.f` to itself. Assume that this method can be analyzed precisely by the base analysis, and that the extended analysis treats it as a difficult method (e.g., because it is a library method). The base analysis without any limits would result in the formula shown to the right of Statement 5. Propagation from this formula eventually results in invalidation of all disjuncts, because the part of the program that precedes this statement ensures that `a.f.g` is *null* only when `y` is also *null*. Therefore, the root dereference is declared safe. However, when the extended analysis uses the immediate producer approach on the predicate $\langle \texttt{z.f.g} = \textit{null} \rangle$ at the point above Statement 8, this predicate gets carried directly to Statements 2 and 4, and then validated at Statement 2. Thus, the root dereference is declared unsafe. This imprecision results because Statement 2 is actually not an immediate producer of `z.f.g` at Statement 8 at all; however, this could have been determined only by a *path-sensitive* immediate producers computation, which can potentially be expensive. We hypothesize (without a proof) that if immediate producers can be computed precisely (using a path-sensitive) analysis then our extended analysis is never less precise than the base analysis with limits.

3.5. Comparison with a simple alternative approach

Sridharan et al. [19] introduced the notion of a *thin slice* of a program wrt a criterion statement. This kind of slice consists of all the *producer* statements of the criterion, as defined in Section 3.1. Sridharan et al. originally proposed thin slicing to support program understanding and debugging tasks; they argued that thin slices were much smaller than full slices, while managing to include statements that are pertinent to program understanding and debugging tasks. Then, subsequently proposed approaches [11, 10, 9] have used the thin slice as a program *abstraction* (as opposed to using the full program) to carry out different kinds of analyses (not null-deference analysis, though). We did originally consider an approach similar to the ones referred to above. The approach is to simply take a thin-slice from the root dereference, and see if any of the statements in the slice is a null-assignment. This approach is sound. It is expected to be less precise than our approach, because while we use immediate producers only when faced with difficult constructs and use a flow-sensitive approach otherwise, this approach uses immediate producers all the time. For instance, in the example in Figure 11(b), say the difficult construct in line 7 is not present. In our approach the immediate producers extension does not kick in; therefore, the base analysis is able to prove the root dereference safe. However, the thin-slice-based approach finds Statement 2 to be in the thin slice, and hence calls the root dereference unsafe.

We have implemented the thin-slice-based approach also, and experimented with it. We present the results in detail in Section 6.5. The summary is that the thin-slice-based approach is less precise than our approach on eight out of ten benchmarks programs, and often, significantly so. The more surprising result is that it is also much less efficient than our approach. It turns out that the primary reason for this is that on a vast majority of dereferences our approach is able to prove them safe within a very small number of propagation steps. Whereas, a thin slice from a dereference usually contains a larger number of producers, which all need to be visited as part of the approach. In other words, flow-sensitivity and precision also result in quick verification.

A related idea is to run our (flow-sensitive) analysis on a (normally) sliced version of the program, rather than on the whole program. One would expect our analysis to run faster, because a sliced program is generally smaller than a full program. However, it was observed [4] that the time saved by performing the analysis on the sliced program did not compensate the time spent in computing the slice in the first place.

4. Java collections and maps

The Java standard library provides built-in implementations of *collections* such as sets, lists and vectors, which are used extensively by Java programmers. Henceforth, we will refer to the API methods in these libraries as *Java collections methods*. When the base analysis enters and analyzes Java collections methods, it ends up analyzing code that accesses complex data structures that are used in the implementations of these collections. Examples of these complex data structures are arrays and recursive data structures. The base analysis does not model array accesses; by applying transfer function GetArray (in Figure 3), it reduces any predicate that contains an access path involving an array reference to *true*. The base analysis cannot reason about the contents of recursive data structures precisely either, because it drops predicates that contain repeated fields (see Section 2). Therefore, entering and analyzing Java collections methods typically does not improve the precision of the base analysis, and yet, adds a lot to the running time. Our objective in the extended analysis is to side-step this difficulty altogether. The approach we take is: (1) Introduce *special fields* in collection objects to model the contents of these collections; these fields will occur in access paths in the formulas that we propagate during the analysis, but are not present in the actual implementations of the collections. (2) Provide manually constructed *summary functions* for all Java collections methods. Each summary function is used as a backwards transfer function; it recognizes and manipulates the special fields, and computes an over-approximation of the weakest-atleast-once precondition before a call-site to the corresponding method given a post-condition after the call-site. When a call to a Java collections method is encountered in the analysis we use the summary function of the method rather than enter and analyze it. This enhances the precision of the analysis significantly, while also improving its efficiency.

Our primary design objective is that the analysis should scale to large, real world Java programs (like our benchmark programs), and have quick response time for verifying a dereference (which is expected from a demand-driven analysis). Therefore, we have chosen to encode *all* collections, including ordered collections such as lists and vectors, simply as unordered sets, and allow only a simple form of existential quantification over the elements of collections in the formulas.

The primary focus of our work is handling Java collections. We also have a simple way to handle *maps*, which we describe at the end of this section.

4.1. The special fields elem and collection

The grammar for access paths that we use in the lattice of the extended analysis, which allows for two special fields, namely *elem* and *collection*, is shown in Figure 12. The rest of the grammar describing the syntax of formulas

$$\begin{array}{lcl} \textit{AccessPath (AP)} & \rightarrow & \textit{Variable.Fields} \mid \textit{Variable} \\ \textit{Fields} & \rightarrow & \textit{field.Fields} \mid \epsilon \\ & & \mid \textit{elem.Fields} \mid \textit{collection} \\ & & \mid \textit{collection.elem.Fields} \end{array}$$

Figure 12: Our structure of formulas (lattice elements) to handle collections

remains the same as in Figure 2. The two special fields have the following meanings.

- *elem*: If u is an access path that points to a collection object, then the access path *u.elem* refers to *any* of the elements stored the collection. Therefore, for instance, the predicate $\langle u.elem.f = null \rangle$ asserts that there *exists* an element in the collection pointed to by u whose field f is *null*. Similarly, the predicate $\langle u1.elem = u2.elem \rangle$ asserts that some element of the collection pointed to by *u1* is the same as some element of the collection pointed to by *u2*. Note that if application code contains references to structures that are part of a collection's internal implementation (which is not common) then these will be represented using normal access paths (i.e., not involving the special fields).
- *collection*: If u is an access path that points to an iterator object, then *u.collection* is an access path that points to the collection object whose elements this iterator refers to (i.e., iterates over).

When we analyze well-typed Java programs using our summary functions for collections methods the access paths in the formulas that arise will necessarily respect the following constraints:

i) An access path preceding an *elem* field can only refer to a collection object.

ii) An access path preceding a *collection* field can only refer to an iterator object. Also, any occurrence of the field *collection* in an access path is either followed by no field, or is followed by *elem*.

The simplification rules in Figure 4 are still applicable even in the presence of the special fields, but with a couple of changes: Rules 2 and 3 are not applicable if the access paths mentioned contain *elem* fields. Also, we use Rule 1 even when the predicate is of the form $\langle ap_1 = ap_2 \rangle$, where ap_1 and ap_2 are syntactically different, in case ap_1 or ap_2 involves the *elem* field. This is because reducing a predicate to *false* can be done only when there is certainty about this, whereas reducing to *true* just results in an over-approximation.

4.2. Overview of our transfer functions

Our objective is to provide summary functions, which we also call *transfer functions* or *rules*, for the methods in the Java Standard Collections API. These

	Container operation	**Transfer Function**: $\lambda\phi \in Disjunct.\phi'$, where $\phi' \in 2^{Disjunct}$, and is =
CollectionAdd	$c.add(v)$	T,

where pseudo-code for computing T is as follows.

```
 1: $T = \{\phi\}$
 2: Let $SubAPs^*(\phi)$ be the set $\{(ap_i.elem, j) \mid ap_i.elem \in SubAPs(\phi),\ 1 \leq j \leq n$, where $n$ is the number of occurrences of $ap_i.elem$ in $\phi\}$.
 3: for all $(ap_i.elem, j)$ in $SubAPs^*(\phi)$ do
 4:     $S = T$
 5:     if $MustAlias(ap_i, c)$ after "$c.add(v)$" then
 6:         for all $\phi_1 \in S$ do
 7:             $T = T - \{\phi_1\} \cup \{\phi_2\} \cup \{\phi_3\}$, where
 8:               $\phi_2 \equiv \phi_1[v/(ap_i.elem, j)] + \{c = ap_i\} + \{c \neq null\}$,
 9:               $\phi_3 \equiv \phi_1 + \{c = ap_i\} + \{c \neq null\}$.
10:     else if $MayAlias(ap_i, c)$ after "$c.add(v)$" then
11:         for all $\phi_1 \in S$ do
12:             $T = T - \{\phi_1\} \cup \{\phi_2\} \cup \{\phi_3\} \cup \{\phi_4\}$, where
13:               $\phi_2 \equiv \phi_1[v/(ap_i.elem, j)] + \{c = ap_i\} + \{c \neq null\}$,
14:               $\phi_3 \equiv \phi_1 + \{c = ap_i\} + \{c \neq null\}$,
15:               $\phi_4 \equiv \phi_1 + \{c \neq ap_i\} + \{c \neq null\}$.
16:     else {$ap_i$ and $c$ are definitely not aliased}
17:         for all $\phi_1 \in S$ do
18:             $T = T - \{\phi_1\} \cup \{\phi_2\}$, where
19:               $\phi_2 \equiv \phi_1 + \{c \neq ap_i\} + \{c \neq null\}$.
```

CollectionAddAll	$v.addAll(c)$	*Treat this as* $v.add(c.elem)$.
ListGet	$v = c.get(i)$	$\phi[c.elem/v]$
GetIterator	$i = v.iterator()$	$\phi[v/i.collection]$
IteratorNext	$v = i.next()$	$\phi[i.collection.elem/v]$
CollectionClear	$v.clear()$	$\phi[false/pred_1][false/pred_2]\ldots[false/pred_n]$, where the $pred_i$'s are the predicates in ϕ such that $v.elem \in SubAPs(pred_i)$.
Init	$v.\langle init\rangle()$	*Treat this as* $v.clear()$.
ParameterisedInit	$v.\langle init\rangle(c)$	$\phi[c.elem/v.elem]$,
Remove	$v.remove(w)$	ϕ
ToArray	$v.toArray()$	$\phi[true/pred_1][true/pred_2]\ldots[true/pred_n]$, where the $pred_i$'s are the predicates in ϕ such that $v.elem \in SubAPs(pred_i)$.

Figure 13: Transfer Functions for Java collections API methods

are the methods that are declared in the interface `java.util.Collection`, as well as the ones declared in its subinterfaces `java.util.Set` and `java.util.List`, as well as methods not listed in these interfaces that are provided specifically by classes that inherit from these interfaces (e.g., the method `getLast` in the class `LinkedList`). We present a selection of our summary functions in Figure 13. There are 23 other summary functions that we have provided in our implementation, which we have not included in Figure 13, which are similar in spirit to the

```
 foo() {
1: u = new Integer(1);     false
2: v = new Integer(2);     ⟨ u = null ⟩
3: z = new HashSet();      ⟨ u = null ⟩, ⟨ v = null ⟩
4: z.add(u);    ⟨z.elem = null⟩,   ⟨ u = null ⟩,  ⟨ v = null ⟩
5: z.add(v);               ⟨ v = null ⟩, ⟨ z.elem = null ⟩
6: i = z.iterator();       ⟨ z.elem = null ⟩
7: w = i.next();           ⟨ i.collection.elem = null ⟩
8: w.toString();           ⟨ w = null ⟩
}
```

Figure 14: Example to illustrate new transfer functions to handle collections. Each entry on the right-hand side shows the formula at the program point that precedes the corresponding statement.

ones in this figure. There remain a few other infrequently used Java Collections methods for which we do not provide summary functions. When our analysis encounters calls to these methods we skip these calls conservatively by simply dropping predicates that involve either of the two special fields and transmitting the remaining predicates to the pre-condition.

Note that at any call-site to a Collections API method we first check (using may points-to information) whether all candidate targets of the call-site are definitely within the standard library implementations in the Java JDK. If yes we use the corresponding summary function (or skip the call conservatively, as mentioned above); else, we first drop all predicates in the post-condition that refer to the special fields *and then* handle the call as we would handle any other call.

Before discussing the details of our transfer functions, we first intuitively illustrate our approach using the complete example in Figure 14. In the example we wish to verify the dereference of `w` at Statement 8. This dereference is safe; this is because `w` points to some element of the collection `z`, and the objects added to `z` (at Statements 4 and 5) are non-null. Note that we abuse terminology, and simply use "the collection `z`" to mean "the collection object pointed to by the variable `z`". Similarly, we use shorthand for iterators, e.g., "the iterator `i`". The initial post-condition to verify the root dereference is $\langle \texttt{w} = \mathit{null} \rangle$ at the point just above Statement 8. At Statement 7, `w` is assigned to an element fetched by the iterator `i`. Unlike in a forward analysis, in our backward analysis we do not know at this point the collection object that `i` refers to; even if we knew this, we would not know the elements that were added to this collection in the preceding code. Therefore, we replace `w` in the post-condition with $\texttt{i}.\mathit{collection}.\mathit{elem}$, yielding the pre-condition $\langle \texttt{i}.\mathit{collection}.\mathit{elem} = \mathit{null} \rangle$, which means *some* element of the collection currently referred to by the iterator `i` is null. Statement 7 is processed by the transfer function IteratorNext in Figure 13.

At Statement 6 it gets revealed that `i`'s iterator refers to the collection `z`; therefore, in our predicate, we replace $\texttt{i}.\mathit{collection}$ (the collection referred to by `i`) with `z`, resulting in the condition $\langle \texttt{z}.\mathit{elem} = \mathit{null} \rangle$. This is taken care of by the GETITERATOR transfer function. At Statement 5, we generate two disjuncts in the precondition: (1) v = *null*, reflecting the hypothesis that the element referred to by z.*elem* in the post-condition is exactly the element pointed to by v, which is being added to z, (2) z.*elem* = *null*, reflecting the hypothesis that the element referred to in the post-condition is *not* the element pointed to by v. This is implemented in the transfer function COLLECTIONADD. Statement 4 is handled similarly. At Statement 3 the variable z is made to point to a new, empty collection. Therefore, no predicate that involves the access path z.*elem*, which refers to the elements in the collection, can be *true* just after this statement. Hence, the transfer function INIT reduces the disjunct $\langle$z.*elem* = *null*$\rangle$ to *false*. The remaining two disjuncts, namely, $\langle$u = *null*$\rangle$ and $\langle$v = *null*$\rangle$, get invalidated at Statements 1 and 2, respectively. Thus, our analysis proves that the dereference at Statement 8 is safe.

4.3. Discussion on the transfer functions

<c.*elem* = *null*>	<c.f = *null*>
v = c.get(i);	**v = c.f**
<v = *null*>	<v = *null*>
(a)	**(b)**
<v = *null*, c = y>, <y.*elem* = *null*, c = y>, <y.*elem* = *null*, c $\neq$ y>	<v = *null*, c = y>, <y.f = *null*, c $\neq$ y>
c.add(v)	**c.f = v;**
<y.*elem*= *null*>	<y.f = *null*>
(c)	**(d)**

Figure 15: Illustration of the similarities between the transfer functions of (a) LISTGET and (b) GETFIELD, and (c) COLLECTIONADD and (d) PUTFIELD.

We now discuss the details of some of our collections-related transfer functions. The COLLECTIONADD rule is the most complex one. It bears resemblance to the PUTFIELD rule in Figure 3. The COLLECTIONADD rule is best understood by comparing it to the PUTFIELD rule, which is what is done in Figure 15, parts (c) and (d). Note that there are two differences between these two rules. The first is that in the PUTFIELD rule we produce two disjuncts, one for the 'c = y' hypothesis and the other for the 'c $\neq$ y' hypothesis (see Figure 15(d)). However, with the COLLECTIONADD rule (see Figure 15(c)), even if c could be equal to y, the new element being added (pointed to by v) may or may not be equal to the element being referred to by y.*elem*. Therefore, we produce a total of three disjuncts in the pre-condition. (In case c and y are must-aliased, which would happen if they are the same variable, then the PUTFIELD rule would produce one disjunct in the pre-condition. Rule COLLECTIONADD would still produce two disjuncts, corresponding to the first two disjuncts shown in the pre-condition in Figure 15(c)).

The second difference comes up when there are multiple predicates in the post-condition involving an access path y.f (which actually does not happen in

Figure 15). With the PUTFIELD rule, in the disjunct in the pre-condition that contains '`c` = `y`', we would replace all these occurrences with the variable `v`. However, if the post-condition after a call to `add` contains multiple occurrences of `y`.*elem*, each of these may independently refer to or not refer to the element `v` being added. In general, if there are k occurrences of the *elem* field in the disjunct in the post-condition, we produce 2^k disjuncts in the pre-condition. In contrast, the PUTFIELD produces 2^m disjuncts in the pre-condition, where m is the number of *syntactically distinct* access paths ending with "$.f$" in the post-condition, irrespective of how many times each such access path occurs in the post-condition.

The COLLECTIONADD rule is described in detail using pseudo-code in Figure 13. Note the utility function $SubAPs^*(\phi)$ used in this rule (see Line 2). This function returns pairs of the form (ap_i, j), which denotes the jth occurrence in the post-condition of the sub-access-path ap_i. For instance, if ϕ is $\langle \texttt{x}.elem.\texttt{f} = null, \texttt{x}.elem = \texttt{y}.elem\rangle$ then $SubAPs^*(\phi)$ is $\{$ $(\texttt{x}.elem, \texttt{1})$,$(\texttt{x}.elem, \texttt{2})$, $(\texttt{y}.elem, \texttt{1})\}$ (while $SubAPs(\phi)$, as defined in Section 2.1.1, would be the set $\{\texttt{x}, \texttt{x}.elem, \texttt{x}.elem.\texttt{f}, \texttt{y}, \texttt{y}.elem\}$). Basically, for each *subset* of the set $SubAPs^*(\phi)$ we generate a disjunct in the pre-condition, obtained by replacing sub-access paths in ϕ that are also in this subset with the variable v (v being the argument passed to the call to *add*). A related point about notation: we use $\phi[v/(ap_i, j)]$ (see Lines 8 and 13) to denote the disjunct obtained by replacing the jth occurrence of ap_i in ϕ with v.

The statement $v.addAll(c)$ adds all elements in collection c to collection v. The corresponding transfer function COLLECTIONADDALL rule is implemented by basically invoking the COLLECTIONADD transfer function as a subroutine.

The statement '$v = c.get(i)$' retrieves the ith element of the list c, and copies it into v. Since in our backwards analysis we do not know the elements that were added to c in the preceding code, in the corresponding rule LISTGET we simply replace occurrences of v in the post-condition with $c.elem$. Note that since we do not keep track of the order of elements inside a list, or even *distinguish* them, the transfer function ignores the index i. Just like rule COLLECTIONADD resembles the PUTFIELD rule in Figure 3, the LISTGET rule resembles the GETFIELD rule in Figure 3. This similarity is illustrated in Figure 15, parts (a) and (b).

Rules GETITERATOR and ITERATORNEXT in Figure 13 are intuitive, and were illustrated in the example in Section 4.2. In fact, the principle behind rule GETITERATOR is similar to the one behind rule LISTGET.

The COLLECTIONCLEAR rule is for modeling the statement '$v.clear()$', which removes all elements from the collection v. The rule invalidates predicates in the post-condition that refer to elements of the collection being cleared. This is a sound thing to do, because an empty collection cannot possibly satisfy any predicate. The rule INIT, which models statements that create a new, empty collection, does the same thing. The rule PARAMETERISEDINIT, which handles the copying of one collection (c) to another (v), is straightforward.

The rule REMOVE is actually used to model three (related) API methods: $v.remove(w)$, $v.removeAll(c)$, and $v.retainAll(c)$. The first of these methods removes the element pointed to by w from collection v; the remaining two methods remove all elements from collection v that are present (resp. not present) in c. In all these cases, removal of an element x really means that *all* objects that are `equals` to the object pointed to by x are removed, where `equals` could be a type-specific user-defined function. Modeling *remove* statements soundly and precisely therefore requires precise analysis of these `equals` methods, which may not be feasible in our demand-driven setting. Therefore, we let REMOVE be an identity function. By choosing to not track removal of elements from collections we basically over-approximate the elements that are treated as being present in any collection at any point. Such over-approximation is sound in our setting; this is because the satisfiability of our formulas (see Figure 12) depends only on what elements are present in a collection, and not on what elements are *not* present in a collection.

In rule TOARRAY, we *drop* all predicates that contain access paths of the form $v.elem$. This is a conservative action we take, because $toArray()$ returns an array of elements, which we do not model.

Appendix B contains proofs of correctness of four key transfer functions in Figure 13. Our approach in these proofs is to first specify the (idealized) concrete semantics of the four corresponding API methods based on the documentations of these methods in the interfaces in which they are introduced. We then prove that our summary functions are over-approximations of the idealized concrete semantics. In this paper we assume that the various actual implementations of these methods in the Java JDK libraries are bug free and faithful to their corresponding idealized concrete semantics. This is also the approach taken by most previous researchers that have proposed to use summary functions of library methods to analyze client code. Interesting (and challenging) topics for future work would include the development of approaches to automatically check whether any given implementation of an interface method is faithful to its idealized concrete semantics, or to extract idealized concrete semantics directly from a method implementation in a sound manner.

4.4. Handling Maps

We also handle the methods in the Java standard library interface `java.util` `.Map` in a simple, conservative way. A map is a collection of (*key*,*value*) pairs. We abstract away the correlation between keys and values, and instead model each map `m` as two collections, `m`.*keys* (its set of keys), and `m`.*values* (its set of values). Note that, *keys* and *values* are special fields introduced only for the purpose of the analysis, like *elem* and *collection*. We treat the operation `m.put(k, v)` as a sequence of two *add* operations: `m`.*keys*.`add(k)` and `m`.*values*.`add(v)`. Note that as per the specified semantics of maps the operation `m.get(k)` returns null when the key referred to by `k` is not present in `m`. Therefore, we translate operation "`m.get(k)`" as "(...) ? `map.values.get(k)` : *null*", where "(...)" is a non-deterministic condition. For example, consider a postcondition $\texttt{v.f} = null$ at the point after the statement $\texttt{v} = \texttt{m.get(k)}$. The precondition will contain two disjuncts: $\langle \texttt{m}.values.elem.\texttt{f} = null\rangle$, considering the non-deterministic condition to be *true*, and $\langle null.f = null\rangle$, considering the non-deterministic condition to be *false*. The latter disjunct reduces to *false*, because *null* does not have any fields (this reduction is accomplished by a simplification rule that will be introduced in 5). The other methods in the `Map` interface are handled in a similar way, conservatively.

5. Implementation Details

5.1. Analysis Framework

We have implemented our approach using the Wala [20] program analysis framework. Wala provides us control-flow graphs of methods, may-points-to information, as well as the immediate-producers of all access paths at all program points (following Sridharan et al.'s approach). All this information is pre-computed by Wala as part of its pre-processing, before our actual analysis starts. We use a flow-insensitive and receiver-type context-sensitive points-to mode of Wala analysis (namely, `ReceiverType ContextSelector`), where the context is based on the concrete type of the receiver object. Our approach uses Wala's points-to analysis results for four purposes: (a) to construct a call-graph (particularly, to resolve virtual method calls), (b) to compute Mod-Ref sets (i.e., side-effect information) for methods, (c) in the PUTFIELD rule (see Figure 3, to reduce the number of aliasing combinations that need to be considered, (d) in the COLLECTIONADD and COLLECTIONADDALL rules (see Figure 13), for a similar purpose as in the PUTFIELD rule, and (e) implicitly during Wala's pre-processing to compute immediate producers of access paths. Imprecision in Wala's points-to analysis affects the precision as well as scalability of our approach due to reasons (a) through (e) above. Our approach would benefit significantly both in terms of precision and scalability from a more precise points-to analysis. However, due to scalability limitations of Wala's points-to analysis implementations, we chose a points-to analysis method that is reasonably precise but highly scalable.

Our implementation is available in open-source form at `http://sourceforge.net/p/npedetector`. Our implementation is single threaded. We analyze every dereference using a separate instance of our analysis; there is no analysis information shared across the verification of different dereferences, except the pre-computed call graph, *may points-to*, and *mod-ref* information. Although sharing of intermediate results within our analysis is certainly possible, and would be beneficial, we avoid this in our experiments in order to estimate running time in a worst-case scenario where only a few selected dereferences need to be verified on-demand. Finally, as with many previous related techniques, our approach does not model concurrency soundly, nor dynamic features such as reflection and dynamic class loading. Also, there are some libraries, for instance, GUI libraries and JDBC libraries, which upon linking cause Wala's may-points-to information computation to become unscalable. Therefore, we omit any analysis of these libraries, and process calls to these libraries using a heuristic that is conservative in most but not all cases.

In the rest of this section we describe a few interesting details about our implementation.

Name	**Instruction**	**Transfer Function**: $\lambda\phi \in \mathit{Disjunct}.\phi'$, where $\phi' \in 2^{\mathit{Disjunct}}$, and is =
s_i: COPY	$v = w$	$U(\phi,\ v,\ s_i)[w/v]$, where 'U' is shorthand for *UpdateOriginatingStmt*
s_i: GETFIELD	$v = r.f$	$U(\phi,\ v,\ s_i)[r.f/v] + \{r \neq \mathit{null}\}$
s_i: PUTFIELD	$r.f = v$	$\phi[r.f, v, ap_1.f][r.f, v, ap_2.f]\ldots[r.f, v, ap_n.f]$, where $\{ap_1.f, ap_2.f, \ldots, ap_n.f\}$ are the access paths in $\mathit{SubAPs}(\phi)$ that end with field f, and $\phi[r.f, v, ap_i.f]$ $= U(\phi,\ ap_i.f,\ s_i)[v/ap_i.f] + \{r \neq \mathit{null}\}$, if $\mathit{MustAlias}(r, ap_i)$ after $r.f = v$ $= \{\, U(\phi,\ ap_i.f,\ s_i)[v/ap_i.f] + \{r = ap_i\} + \{r \neq \mathit{null}\},$ $\phi + \{r \neq ap_i\} + \{r \neq \mathit{null}\}\}$, if $\mathit{MayAlias}(r, ap_i)$ after $r.f = v$ $= \phi + \{r \neq \mathit{null}\}$, otherwise

Figure 16: Modified transfer functions for tracking *originating* statement

Algorithm: *UpdateOriginatingStmt*
Input: a disjunct ϕ, access path *ap*, and statement s_i
Output: a disjunct ϕ', with the originating statement of access path *ap* updated to s_i

```
Step 1: $\phi' = \phi$
Step 2: FOR ALL $(ap, s_j)$ in $APs(\phi')$
Step 3:   Replace $(ap, s_j)$ in $\phi'$ with $(ap, s_i)$
Step 4: END FOR
Step 5: RETURN $\phi'$
```

Figure 17: Pseudocode to update originating statement of an access path

5.2. Determining immediate producers

Wala's immediate-producers API actually provides the immediate producers of any given *statement.* However, as discussed in Section 3.2, we require in our approach a routine *ImmProds(ap,p)* which returns the immediate producers of an access path *ap* at a program point *p*. The immediate producers of a statement are nothing but the immediate producers of access paths that are used in the statement for purposes other than pointer dereferencing. For instance, the immediate producers of a statement `u = v.g` are the immediate producers of the access path `v.g` (but not the immediate producers of `v`) at the point that precedes this statement.

Our approach to implementing the routine *ImmProds* using Wala's API call is as follows. We associate with every access path in a formula a statement, which we refer to as its *originating* statement, which is the statement at which the access path was *originally* introduced into the formula. The idea is that the immediate producers of an access path in a formula at any program point are nothing but the immediate producers of the originating-statement associated with that access path.

In order to track the originating statements of access paths we extend the syntax of access paths that we use in formulas, and let each access path be a pair of the form (*Variable.Fields, stmt*) or (*Variable, stmt*). Then, we modify the transfer functions GETFIELD, PUTFIELD, and COPY (see Figure 3) to incorporate originating-statement information into access paths. Figure 16 shows these modified transfer functions, while Figure 17 shows the accessory function *UpdateOriginatingStmt* used in these transfer functions to update the originating statement of an access path. The function $APs(\phi)$ used in *UpdateOriginatingStmt* returns only the (complete) access paths (and not the extended sub-access paths) that occur in ϕ. For instance, $APs((v.f, s_n) = \mathit{null})$ will only return $(v.f, s_n)$ and not (v, s_n). Note that, as per our definition of algorithm *UpdateOriginatingStmt*, we replace the originating statement of an access path *ap* with a statement s_i only when the transfer function of statement s_i replaces *ap entirely* with a different access path. For instance, for a copy instruction $s_i : v = w$, and if the postcondition is $\langle (v, s_j) = \mathit{null}, (v.f, s_k) = \mathit{null} \rangle$, the precondition will be $\langle (w, s_i) = \mathit{null}, (w.f, s_k) = \mathit{null} \rangle$. This makes sense, because the immediate producer of *w.f* at the point preceding s_i is nothing but the immediate producer of *v.f* at the point after s_i.

There arise situations where an access-path has no originating statement associated with it. For instance, a statement of the form $v = c.get(k)$, where c is a collection, will introduce an access path *c.elem* into its pre-condition; this access path cannot have an originating statement, because the access path uses a special field (which Wala's thin slicer will not recognize). Whenever an access path does not have an associated originating statement we will essentially be unable to find its immediate producers, and hence cannot skip a difficult construct.

Another noteworthy aspect is that fields of newly created objects are by default assigned to *null* in Java. The Wala IR does not represent this *implicit null assignment*. Therefore, if a field f of an object pointed to by a variable v is never explicitly assigned on a path from the program's entry to a point p, then *ImmProds(v.f, p)* will miss the *null* that implicitly flows to *v.f* at p, hence missing an immediate producer. This would make our analysis unsound. Therefore, before using the immediate producers returned by Wala of an access path *ap.f* at a point p, we check if *ap.f* has been assigned on all paths to p. We do this using an inexpensive limited-scope forward analysis from the *new* statements that create the objects that *ap* may point-to. If within this limited-scope analysis we are not able to confirm that f is definitely assigned a value, we conservatively assume that the immediate producers of *ap.f* returned by Wala are unsound, and do not employ our extension to skip the difficult construct.

5.3. New simplification rules to invalidate infeasible disjuncts

We use several additional simplification rules in our extended analysis over the base rules shown in Figure 4, in order to invalidate *infeasible disjuncts*, which are disjuncts that flow along infeasible paths. This increases the precision of our analysis.

- Say method m of class C is being analyzed and a disjunct at some point in m contains the predicate $\mathit{this}.f = \mathit{null}$. It may so happen (due to imprecision in Wala's points-to analysis and call graph) that the class in which field f declared is neither C nor any superclass of C. In such a case we invalidate this predicate, because the access path $\mathit{this}.f$ is infeasible.
- If an access path $\mathit{null}.f_1.f_2.f_3 \ldots f_n$ occurs in a disjunct, where $n \geq 1$, we invalidate the disjunct. This is because *null* does not have any fields.
- If an access path $t_i.f_1.f_2 \ldots f_n$ occurs in a disjunct, where $n > 1$, and t_i is a variable representing a newly created object (see rule NewAsgn in Figure 3), we invalidate the disjunct. This is because f_1, being a field of a newly created object, is implicitly *null*.
- If a disjunct ϕ containing an access path $\mathit{ap.elem}$ reaches the entry of the program then this disjunct is invalidated. This is because it is impossible for any collection to contain any elements at the beginning of a program.

6. Experimental Results

In this section we address the following research questions by empirical evaluation using our implementation:

RQ1 How many null-dereference warnings does the base analysis produce, and is it scalable?

RQ2 Is the extended analysis more precise than the base analysis? Is it also scalable?

RQ3 What is the contribution of our immediate producers idea (refer Section 3) to the gain in the precision of the extended analysis over the base analysis?

RQ4 What is the contribution of our idea of handling Java collections using summary functions (refer Section 4) to the gain in the precision of the extended analysis over the base analysis?

RQ5 Is the extended analysis more precise than the simple thin-slice based approach discussed in Section 3.5? Which one is more expensive?

6.1. RQ1. How many null-dereference warnings does the base analysis produce, and is it scalable?

We used 10 real-world Java programs for all our primary experiments discussed in this section, key information about which is presented in Figure 18. In each benchmark we considered all the methods that are reachable along the call-graph from all the *main* methods, excluding main methods in test-suites,

Benchmarks	Analyzed Appl. bytecodes (1)	Analyzed Lib. bytecodes (2) Base (a)	Extended (b)	No. of appl. methods (3)	Total # derefs (4)	# unsafe derefs by base analysis (5)	# unsafe derefs by extended analysis (6)
bcel 5.2	86483	3596	28728	10092	10143	1221	653
jbidwatcher 2.1.2	105043	16505	112702	4076	9643	1652	769
javacup 0.1	29180	509	29563	70548	2851	608	186
sablecc 4.2	157169	4055	16389	28821	14017	2139	1220
jlex 1.2.6	25056	408	1329	1264	2510	93	28
l2j 3.7	373661	14310	68565	591	36899	6015	4100
proguard 4.5	185594	2766	33474	19195	18275	2781	1789
ourtunes 1.3.3	127167	7206	34639	162	16449	1532	1029
antlr 3.3	251010	7224	30317	14472	17409	4042	2072
freecol 0.9.5	260785	8889	113012	16432	24077	6994	4281

Figure 18: Results of base and extended analyses on 10 real world Java programs.

Benchmark (1)	Pre-proc. time (sec) (2)	Running time base analysis (sec) (3)	Running time ext. analysis (sec) (4)
bcel	11	34	190
jbidwatcher	17	470	470
javacup	7	23	68
sablecc	20	74	656
jlex	6	9	13
l2j	23	1159	1125
proguard	19	116	1415
ourtunes	12	203	72
antlr	18	43892	45464
freecol	32	5815	39952

Figure 19: Analysis time of the base and extended analysis on 10 real world Java programs.

as the *scope of analysis*. That is, our analysis is restricted to this scope, and so are all metrics that we present in the rest of this section. Column (1) shows the number of bytecodes in the methods in the actual application portion of the benchmark. Column (2)(a) shows the number of bytecodes in the linked library methods that were actually visited and analyzed by the base analysis, while Column (2)(b) gives the same information wrt the extended analysis. Note that due to the various heuristics and extensions, the library methods that are visited by either of the two analyses mentioned above need not be the same or even be a subset of the ones visited by the other analysis. Column (3) shows the total number of application methods. We treated each dereference in each non-library method, except dereferences to the variable *this* (which are always guaranteed to be safe), as a root dereference to be verified. The total number of such dereferences in each benchmark is shown in Column (4).

We selected these ten particular benchmarks for our primary experimentation because they were used by previous researches working on null-dereference analysis for Java. For instance, we used these same benchmarks in our previous work [4], and before this Loginov et al. [17] used 8 of these 10 benchmarks.

Reasons for Imprecision (1)	Average % over 10 benchmarks (2)		
	Base analysis (a)	Base analysis w/o limits (b)	Extended analysis (c)
null-assignment	19.4	21.2	35.4
unbounded-aps	40	54.3	36
call-backs	10.4	NA	NA
virtual-calls	15.9	NA	NA

Figure 20: Reasons for dereferences being declared unsafe

We carried out our experiments by linking our benchmarks to the Open JDK 1.6 libraries, on a server machine having 2.27 GHz 8-core Intel Xeon processor, with 16 GB RAM.

Throughout this section, unless otherwise noted explicitly, whenever we refer to the base analysis, we mean the base analysis *with* the limits referred to at the beginning of Section 3. Column (5) of Figure 18 shows the number of dereferences (from the total number shown in Column (4)) that were reported as unsafe by the base analysis. The percentage of dereferences reported as unsafe by the base analysis ranges from 3.7% in Jlex to 29% in freecol. The arithmetic mean (resp. geometric mean) of the percentage of dereferences reported as unsafe in each of the ten benchmarks by the base analysis is 16.3% (resp. 14.4%). Note that unfortunately it is not straightforward to determine the percentage of these unsafe reports that are false positives.

Figure 19 shows for each benchmark Wala's pre-processing time for computing may points-to information in Column (2), and the total running time to verify all dereferences in Column (3). Over all ten benchmarks the base analysis takes only 288 millisecs on average to verify a dereference.

6.1.1. Reasons for unsafe dereferences

In order to understand the performance of the base analysis better, we identified four major reasons why it declares dereferences as unsafe, as is listed in Column (1) of Figure 20. The *null-assignment* category includes all the unsafe dereferences during whose analysis a disjunct containing a root predicate was propagated through a null-assignment statement that replaced the access-path in the root predicate by null and hence reduced the root predicate to *true*. For instance, this would happen if a disjunct $\langle v.f = \mathit{null} \rangle$ is propagated thorough the statement "$v.f = \mathit{null}$". For the dereferences that fall under this category there exists at least one static path along which a null value flows to the root dereference (although this path could be an infeasible one, not eliminated by our limited notion of path sensitivity). An unsafe dereference falls in the *unbounded-aps* category if during its analysis the root predicate of some disjunct was reduced to *true* because it had an access path with a repeating field or a reference to an element of an array (see Section 2.1). A dereference falls into the *call-backs* category, if during its analysis some disjunct containing a root predicate reached the entry of a method that is called-back by a library method. The base analysis stops at this point and declares dereference to be unsafe. A dereference falls into the *virtual-calls* category if during its analysis some disjunct containing a root predicate had to be propagated though a virtual method call with more targets than the pre-set threshold, with at least one of targets affecting the root predicate, and the limit in the base analysis (see the beginning of Section 3) reduced this root predicate to *true*.

Column 2(a) in Figure 20 shows, for each reason of imprecision, the average percentage of dereferences across all benchmarks that were called unsafe due to this reason. A single dereference can go into multiple categories as several disjuncts may be introduced at various program points during the analysis of this dereference, with each of them being reduced to true for a different reason. The categories cover the most important reasons for imprecision but not every possible reason. In other words, some dereferences may not fall into any of the categories.

It is clear that the top reason for dereferences being called unsafe are arrays and recursive data structures (i.e., the *unbounded-aps* category). Enhancing our analysis to address these directly and precisely would be expensive, and would likely render our approach not usable in a demand-driven setting with very fast response time. This is why we have chosen to indirectly address this problem using summary functions for Collections API methods, which are typically heavy users of arrays and recursive data structures. The null-assignment category can possibly be shrunk by incorporating more path sensitivity in our analysis; however, this again is likely to impact the efficiency of the analysis. Finally, our immediate producers idea is addressed directly at reducing the negative impact of virtual-calls and call-backs on the base analysis.

6.1.2. Empirical evaluation of the base analysis without limits

We also evaluated the possibility of removing the three limits in the base analysis (see the discussion at the beginning of Section 3). The analysis in this mode does not complete on two of the ten benchmarks, namely, *antlr* and *freecol*, even after 15+ hours of running. It completes on the other eight benchmarks, but is extremely inefficient. When we remove only the limits on virtual calls and library calls, and hence enter and analyze all targets of all virtual calls and all library methods, the analysis on average spends *37 times* more time to verify each dereference. This slowdown is due to the large numbers of candidate targets that some virtual calls have, and also due to the size and complexity of library codes. Yet, the precision gain is very low – the number of unsafe dereferences reported comes down by only 3.4% on average per benchmark. Similarly, removing the limit on propagating disjuncts back from library call-back methods increases the average time to verify a dereference by 3 times, while decreasing the unsafe dereferences reported by only 0.27%. When we remove both the limits the analysis takes *87 times* more time on average to verify a dereference, while reducing the unsafe dereferences reported by merely 4.1%.

One of the main reasons why we did not observe significant precision gain in the experiments above is that when we allow the base analysis to analyze difficult constructs it in many cases ends up encountering either array accesses or recursive data structure accesses. This is evident from the results shown in Column 2(b) of Figure 20. Note that the percentage of unsafe dereferences that fall into the category of *unbounded-aps* has gone up from 40% to 54.3%. Another important reason, which is harder to quantify, is that imprecision in the call-graphs provided by Wala result in many spurious candidate target methods at virtual call-sites. These spurious targets hold back the precision of the analysis because if a disjunct gets validated inside a spurious target method during the analysis of a root dereference then having entered all the (non-spurious) targets and having analyzed them is of no avail.

6.2. RQ2. Is the extended analysis more precise than the base analysis? Is it also scalable?

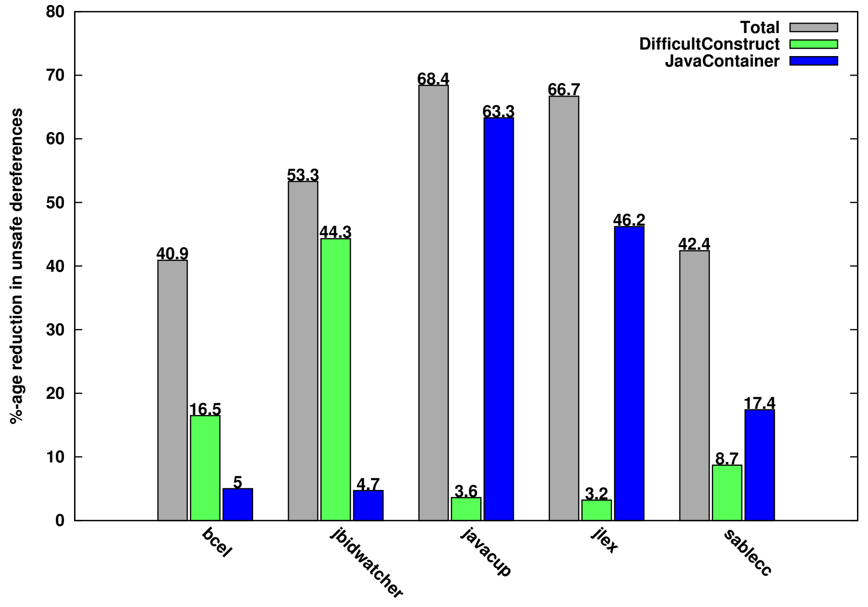
Column (6) in Figure 18 shows the number of dereferences reported as unsafe by the extended analysis (i.e., with both our extensions turned on). The percentage of dereferences reported as unsafe by the extended analysis ranges from 1.2% in Jlex to 17.8% in freecol. The arithmetic mean (resp. geometric mean) of the percentage of dereferences reported as unsafe in each of the ten benchmarks by the extended analysis is 9% (resp. 7.6%), vis-a-vis 16.3% (resp. 14.4%) for the base analysis. Figure 21 shows three bars for each benchmark, with the smaller benchmarks being in part (a) of the figure and the larger benchmarks being in part (b) of the figure. The first bar (i.e., the tallest bar) for each benchmark indicates the percentage reduction in number of unsafe dereferences reported by the extended analysis in comparison to the base analysis; in other words, this is the percentage by which the number in Column (6) of Figure 18 for this benchmark is less than the corresponding number in Column (5). We discuss the other two bars for each benchmark in Sections 6.3 and 6.4, respectively.

It is noteworthy that the extended analysis is also significantly more precise than the base analysis without its limits. The reasons why removing the limits from the base analysis did not improve its precision significantly were discussed in Section 6.1.2.

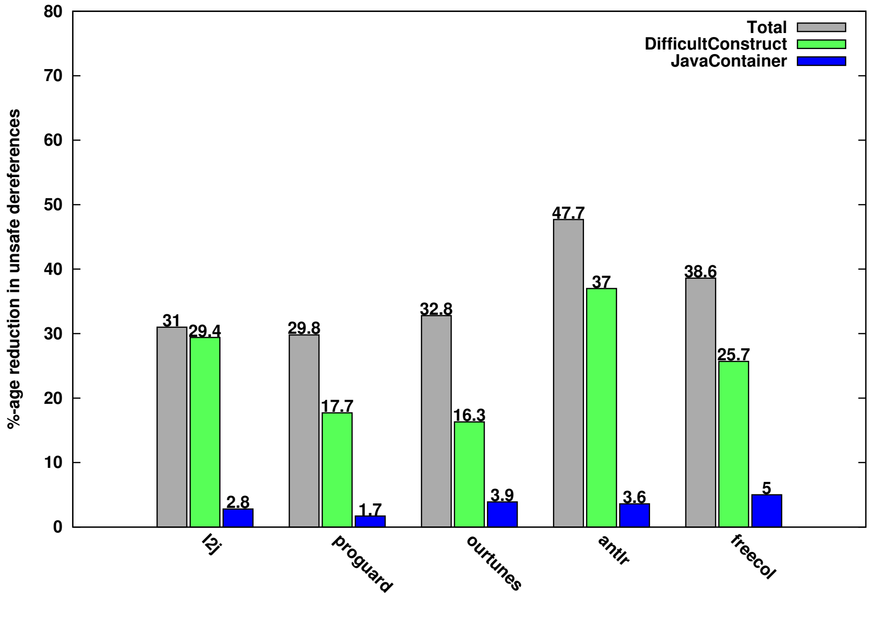
Column 2(c) of Figure 20 shows the reasons for imprecision of the extended analysis. Note that in comparison to the base analysis without limits, whose corresponding numbers are shown in Column 2(b), the incidence of unbounded-aps has come down significantly. The contribution by the null-assignment category has increased a lot, indicating that infeasible paths are potentially a major source of imprecision in the extended analysis. We wish to reiterate, however, that the denominators used in these percentage calculations (i.e., the number of unsafe dereferences) are *different* in these two columns.

6.2.1. Correlation between propagation count and precision

We now discuss a metric about the analysis that correlates to precision: *average propagation count*. The *propagation count* of a dereference refers to the length of the longest path of instructions through which a disjunct is propagated during the analysis of the dereference. We determine this by associating a count with each disjunct; this is set to zero for the original disjunct created at the root dereference. Whenever the transfer function of an instruction generates one or more pre-condition disjuncts from a given post-condition disjunct at



(a) Smaller benchmarks



(b) Larger benchmarks

Figure 21: Reduction in unsafe dereferences (%-age) when compared to the base analysis when the extended analysis is run in different modes.

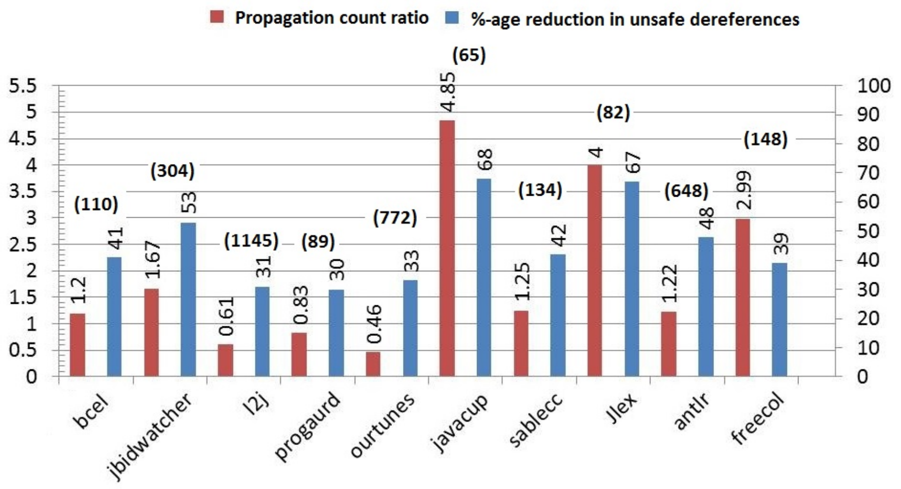


Figure 22: Increase in precision and in propagation count of the extended analysis. The left y-axis indicates the ratio of average propagation count of the extended analysis over the base analysis, while the right y-axis indicates the corresponding reduction in percentage of unsafe dereferences. The number within parenthesis for each benchmark is its average propagation count under the base analysis.

the point after the instruction, the generated disjuncts get a count that is one more that of the post-condition disjunct. The following situations are treated as if they involve propagation through a single instruction: a summary hit from the summary table Σ, a propagation of a disjunct along an immediate producer edge, and the generation of a disjunct by one of our manually constructed summary functions. The propagation count of a dereference, then, is the maximum propagation count of any disjunct that gets generated during the analysis of the dereference. Finally, the average propagation count for a benchmark is the average propagation count of all its dereferences. For each of our ten benchmarks, the first bar in Figure 22 shows the ratio of the average propagation count of the extended analysis over the base analysis. The second bar shows the percentage reduction in unsafe dereferences reported by the extended analysis over the base analysis; here, a taller bar indicates more gain in precision (this bar is a reproduction of the first bar in Figure 21 for the same benchmark).

There are several points to note in these results. (a) In seven out of ten benchmarks the propagation counts are higher in the extended analysis when compared to the base analysis. This result indicates that the situations wherein the *limits* in the base analysis cause it to stop early and give up whereas the extended analysis continues the analysis are frequent. (b) Across all ten benchmarks there is a general positive correlation between change in average propagation count and increase in precision. This confirms our expectation that if we reduce the situations where we give up early and pursue longer paths we get higher precision. (c) In the benchmarks *l2j, proguard* and *ourtunes* the ex-

Benchmarks (1)	Total Deref (2)	Base unsafe (3)	Diff-Constr. Related (4)	Diff-Constr. proved safe (5)	Collections Related (6)	Collections related proved safe (7)
bcel	10143	1221	344	201	191	61
jbidwatcher	9643	1652	914	731	490	78
javacup	2851	608	24	22	410	385
sablecc	14017	2139	265	186	827	373
jlex	2510	93	10	3	55	43
l2j	36899	6015	2367	1771	235	166
proguard	18275	2781	1243	493	113	46
ourtunes	16449	1532	253	250	97	59
antlr	17409	4042	2495	1496	456	146
freecol	24077	6994	2167	1796	1302	353

Figure 23: Improvement due to our extensions to handle difficult constructs and Java collections

tended analysis has higher precision than the base analysis (albeit, by a smaller margin than in the other benchmarks) even though the propagation count has fallen. For certain dereferences, our extensions actually reduce the propagation count (because, e.g., we do not enter and analyze a library call or targets of a virtual call).

The increase in propagation counts has caused a corresponding increase in running time of the analysis. Column (4) in Figure 19 shows the total time to verify all dereferences in each benchmark by the extended analysis. This is to be contrasted with the corresponding numbers for the base analysis in Column (3). The extended analysis takes, on average, 427 ms to verify a dereference, in contrast with the 288 ms requirement of the base analysis. The time requirement of the extended analysis is still very tolerable, and is justified by its higher precision. Furthermore, we have empirical evidence that the extended analysis spends extra time only on the dereferences that the base analysis calls unsafe; the average analysis time per dereference has increased by only 1% for the dereferences that the base analysis reported as safe. The extended analysis takes more time to verify the dereferences that were called unsafe by the base analysis because the extended analysis continues its work even after the point where the base analysis would have given up.

The pre-processing time for the extended analysis analysis is only one or two seconds more than the pre-processing time for the base analysis, even though we ask for immediate producers to be pre-computed in the extended analysis. Therefore we have not shown pre-processing time for the extended analysis analysis separately.

6.3. RQ3. *What is the contribution of our immediate producers idea (refer Section 3) to the gain in the precision of the extended analysis over the base analysis?*

We now study our immediate producers extension in isolation (i.e., without using the manual summary functions of collections methods). The idea is to bring out the potential applicability of this extension, and also to bring out the extent to which the extension meets its potential. Figure 23 once again lists all our ten benchmarks. Columns (2) and (3) in this figure are basically the same as columns (4) and (5) in Figure 18, respectively. In Column (4) of Figure 23 we report the number of dereferences among the ones in Column (3) that had their *root predicate* reduced to *true* due to the limits imposed by the base analysis for avoiding the analysis of difficult constructs (see the discussion at the beginning of Section 3). These are the dereferences that can potentially benefit from our extension. In Column (5) we report the dereferences among the ones included in Column (4) that are proved safe by our extension. This ratio of Column (5) over Column (3) is shown as a percentage by the second bar for every benchmark in Figure 21.

As evidenced by Columns (4) and (5) in Figure 23, on average across all benchmarks, we are able to prove 68.63% of the dereferences reported in Column (4) as safe. Our difficult-construct extension reduces unsafe dereferences by 20.24% (on average across all benchmarks), while increasing average time to verify a dereference by only 2.5 times. These results clearly show the value of our extension.

The ratio of Column (5) over Column (4), in general, cannot be 100% because a dereference falls into Column (4) if *some* disjunct along some path during the analysis of the dereference encounters a difficult construct. Along other paths other complicating factors (e.g., array accesses, or recursive data structures) may be encountered which the immediate-producers extension does not address, and which cause a disjunct to get validated, hence resulting in the dereference still being called unsafe.

6.4. *RQ4. What is the contribution of our idea of handling Java collections using summary functions (refer Section 4) to the gain in the precision of the extended analysis over the base analysis?*

We now discuss the effect of turning on *only* our extension to handle Java collections methods. Column (6) in Figure 23 shows the number of unsafe dereferences reported by the base analysis during the analysis of which a root predicate was reduced to *true* when the analysis was inside the body of a Java collections method. This typically happens because the implementations of Java collections use complex data structures like arrays or recursive data structures, which are not handled precisely by the base analysis. In Column (7) we report the dereferences among the ones included in Column (6) that are proved safe by our summary-functions based technique to handle Java collection methods. The ratio of Column (7) over Column (3) is shown as a percentage in the third bar for every benchmark in Figure 21. Note that in Figure 21 the third bar is taller that the second bar for three programs – *javacup*, *jlex*, and *sablecc*. This implies that in these three programs our collections summary functions are effective than our immediate producers extension in proving dereferences safe.

Ideally, the numbers in Column (7) should be close to those reported in Column (6). This happens on the benchmarks *jlex*, *javacup*, *ourtunes* and *l2j*, in which on average the number in Column (7) is 76% of the number in Column (6).

Benchmarks (1)	Total Deref (2)	Base unsafe (3)	Collections Related (6)	Collections related proved safe (7)	Extended unsafe (8)
mst	69	15	7	7	3
em3d	69	23	17	17	6
health	111	22	6	2	14
Power	211	10	8	1	2
voronoi	174	61	49	11	50
bh	242	53	26	25	7

Figure 24: Our results on the Jolden benchmarks

In the benchmarks *sablecc*, *proguard*, *bcel*, *antlr*, and *freecol* the impact of our technique is moderate; the numbers in Column (7) are 45%, 40%, 31%, 32% and 27% of the corresponding numbers in Column (6). In the *jbidwatcher*, we do not observe much reduction in the dereferences being reported unsafe earlier. We already discussed in Section 6.3, why *all* the dereferences in a category of imprecision in general cannot be proven safe just by turning on the extension for that category.

On a related note, in general, there are some dereferences that can be verified as safe if *both* of our extensions are turned on simultaneously, but not when either one of the extensions alone is turned on. This happens when different extensions help invalidate different disjuncts during the analysis of the dereference. This is why in Figure 21 the first bar in each benchmark, except *l2j*, is taller (and sometimes significantly so) than the sum of the heights of the last two bars.

We also additionally use the variant of the *Jolden* benchmarks created by Marron et al. [12] to evaluate their approach. The original *Jolden* benchmarks are based on the *Olden* C-language benchmarks, and are pointer-intensive programs. Marron et al. replaced uses of ad-hoc data structures in the original Jolden programs wherever possible with uses of the standard Java collection APIs. Marron et al. also put an extra wrapper API over the Java collections API, which we have removed. We only used 6 of the 9 Jolden benchmarks, as the others did not have any usage of collections. The reason we did this study was that since the Jolden benchmarks do not possess some of the other complications present in real benchmarks, such as virtual calls with too many candidate targets, our results on these benchmarks serve as a sort of limit study on the effectiveness of our summary functions approach. Figure 24 shows the results on each of these benchmarks. Columns (2), (3), (6), and (7) have the same meanings as the corresponding columns in Figure 23. Note that the numbers in Column (7) were obtained with only our extension for Java collections turned on. On benchmarks *bh*, *mst* and *em3d*, our extension to handle Java collections methods has been remarkably effective; the numbers in Column (7) are on average 96% of the numbers in Column (6). Considering all of the six micro benchmarks this number works out to 60%, which is higher than the corresponding number of 49% observed with the real benchmarks. Column (8) shows the number of dereferences that are declared unsafe by the full extended analysis, i.e., by turning both of our extensions on. In the six micro benchmarks

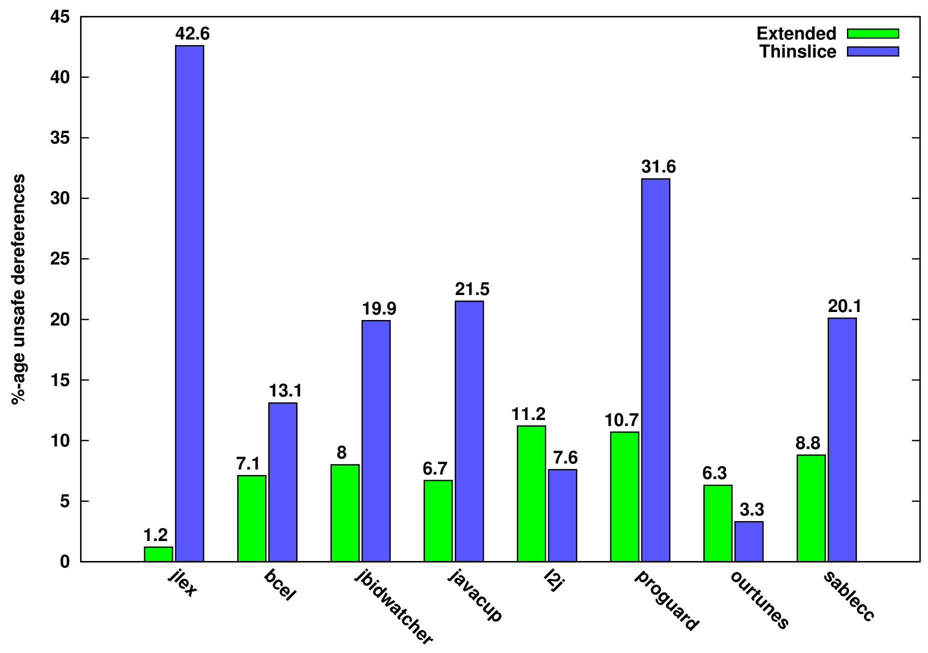


Figure 25: Precision comparison of the thin-slice based approach and the extended analysis

the extended analysis has, on average, proved safe 60% of the dereferences in each benchmark that were declared unsafe by the base analysis; the corresponding number with the real benchmarks was 45%. Thus, our extensions serve their intended objectives well.

6.5. RQ5. Is the extended analysis more precise than the simple thin-slice based approach discussed in Section 3.5? Which one is more expensive?

We will now present the results of our empirical evaluation of the thin-slice based approach discussed in Section 3.5 using an implementation of ours of this approach. This approach did not complete on two of our ten benchmarks, namely, *antlr* and *freecol*, even after running for 15+ hours. Figure 25 shows a comparison of the percentage of the dereferences that were reported as unsafe by the thin-slice based approach versus the results from the extended analysis on the remaining eight benchmarks In 6 out of 8 benchmarks the extended analysis reports much fewer unsafe dereferences than the thin-slice based approach. The average percentage of dereferences reported as unsafe over all the benchmarks is 19.9% for the thin-slice based approach versus 7.5% for the extended analysis.

We now discuss the potential reason why the thin-slice based approach gives better precision than the extended analysis on two of the benchmarks, namely, *l2j* and *ourtunes*. Recall, as per the discussion in Section 5.2, that Wala actually under-approximates the set of immediate producers of any statement. We had discussed in that section a workaround to ensure that our approach is sound in

Benchmarks	Total	Base Analysis (3)		Unsafe derefs (4)	
	derefs	Unsafe Derefs	Time[sec]	Unsafe Derefs	Time[sec]
(1)	(2)	(a)	(b)	(a)	(b)
bloat	18044	3592	311	2228	11342
chart	16139	1972	556	1460	9721
fop	1211	187	11	163	14
hsqldb	829	138	5	95	7
luindex	1957	280	15	236	16
lusearch	3121	696	1727	532	1827
xalan	376	70	4	60	4
eclipse	5713	1242	977	863	2422

Figure 26: Results of the base and extended analyses on the Dacapo benchmarks

spite of this limitation of Wala. We did not implement this same workaround in our implementation of the thin-slice based approach, due to its complexity. Therefore, our implementation of the thin-slice based approach is actually unsound, meaning it can potentially miss null dereferences. If we somehow updated our implementation to use the workaround in order to ensure soundness it is possible that the number of dereferences reported as unsafe will increase significantly.

Contrary to our expectations, the thin-slice-based approach turned out to be also less efficient than our extended analysis. As mentioned above, it did not scale to two of the benchmarks. On the remaining 8 benchmarks it takes on average 337 ms to verify a single dereference in comparison to 47 ms by the extended analysis. The reason for this is that in the version of Wala that we are using (version 1.1.3), when we ask for the immediate producers of a statement, the Wala API method in fact first performs a linear search of the entire dependence graph to locate that statement. The pre-computed immediate producers of this statement are then returned. The above-mentioned linear search is time consuming, and needs to be carried out at every step with the thin-slice-based approach. Whereas, in the extended analysis, we query for immediate producers only at difficult constructs. It would be possible to modify the Wala code to avoid the need for the linear search, in which case the thin-slice-based approach is likely to become more efficient than our approach.

6.6. Experimentation using the Dacapo benchmarks

We have performed our primary experimentation with the ten benchmarks mentioned in Figure 18 for reasons that we mentioned in Section 6.1. However, for additional validation, we have also evaluated our approach on the Dacapo 2006 [21] benchmarks, which have been used by researchers in the past for various kinds of evaluations (although not, to our knowledge, to evaluate verification approaches that are similar to ours).

We could not run our analysis on two Dacapo benchmarks, namely, *pmd* and *jython*, as the base as well as extended analyses both faced out-of-memory issues with these benchmarks. Regarding *antlr*, Dacapo 2006 includes version 2.7.2 of this benchmark, whereas version 3.3 is already included in our primary set of 10 benchmarks. Therefore, we ignore the Dacapo version of this benchmark. Figure 26 shows the results of running the base analysis and the extended analysis on the remaining eight Dacapo benchmarks. For each benchmark we show the total number of dereferences analyzed in Column (2), the number of dereferences reported unsafe and total analysis time taken by the base analysis and the extended analysis in Column (3) and (4), respectively. The total number of dereferences that we analyzed is lower than expected in the benchmarks other than *bloat* and *chart*. One reason for this could be that there are many methods that are called using reflection, the dereferences within which we would not analyze.

The arithmetic mean (resp. geometric mean) of the percentage of dereferences reported as unsafe in each of these eight Dacapo benchmarks by the base analysis is 17.6% (resp. 17.3%). The corresponding figures for the extended analysis analysis are 13.3% (resp. 13%). The base analysis takes on average over all these 8 benchmarks 101 ms to verify each dereference whereas the extended analysis takes 214 ms.

7. Related work

We categorize related work in three ways, and discuss each category separately. The first category is about approaches for null-dereference verification, in general, and how they relate to the base analysis. The other two categories are related to our two extensions, respectively.

7.1. Null dereference analysis

Xylem [16] and *Snugglebug* [2] are two previous approaches that are closely related to the base analysis, in the sense that they compute weakest preconditions in Java programs using a backwards analysis. Xylem is an unsound approach, unlike the base analysis, in that they may miss real bugs. However, several of the design decisions in the base analysis are inspired by Xylem; e.g., demand-driven analysis, use of predicates as data-flow facts, custom simplification rules, etc. Xylem uses a richer set of predicates than our approach, for higher precision. On the other hand, they deal with recursion in an unsound manner, whereas the base analysis computes fix-points. Snugglebug's objective is to try to find a *concrete* input to a program that *disproves* a desired safety property. Their approach is much more expensive than the base analysis. They *under-approximate* the weakest pre-condition, whereas the base analysis over-approximates it. The work of Sinha et al. [22] also proves safety properties by propagating formulas. Their idea of using backwards analysis, with "forward" descent into call targets from call sites is very similar to the *depth-first* propagation that the base analysis employs, as was discussed in 2.3. Additionally, it exploits caller/callee invariants inferred from failures in satisfying formulas to prune future search. Blackshear et al. [23] propose a technique to perform refutation of heap reachability using backward symbolic execution. The objective of this approach is to find memory leaks in Android programs. Like the base analysis, this technique also performs strong updates, and is path- and context-sensitive.

Salsa [17], and approach of Spoto et al. [18] target null-dereference verification of Java programs using a forward analysis; i.e., they are not demand driven. For a more detailed discussion on these approaches we refer the reader to earlier work [4].

Shape analysis [3] is a precise but heavy-weight technique for verifying various properties of heaps. As such shape analysis can be used for null-dereference analysis also. There exist several techniques [24, 25, 26] for performing shape analysis by backward analysis. Of these, only the paper by Gulavani et al. [26] spells out an inter-procedural analysis, and provides empirical evidence. The largest benchmark program they evaluated their approach on has 460 LOC, with the corresponding analysis time being 75 seconds (for the complete program). Among the forward (i.e., non demand-driven) shape analysis techniques, the one due to Yang et al. [27] has been shown to scale to real programs (device drivers) of size approximately 10 kLOC, taking on average 430 seconds to analyze each benchmark.

7.2. Virtual Calls and Call-backs

We now compare our technique of using immediate producers to skip difficult constructs with three previous related approaches that use *thin slicing* for solving verification problems. A *thin slice* from a data reference `r` is basically the set of statements in the backward transitive closure of the *immediate producer* relation (defined in Section 3.1) from data reference `r`. Tripp et al. [10] propose a technique to perform static *taint analysis* on Java programs to detect security vulnerabilities in web applications. Geay et al. [9] perform *static permission analysis* on programs built by assembling components. This is done to find permissions for components such that these permissions are neither too restrictive nor too permissive. Hammer et al. [11] propose an approach to identify potential run-time types of object references. All three techniques use thin slicing to identify all possible *sources* from which information flows (via transitive copy statements) into a given sink. In Section 3.5, we had mentioned how thin slices could potentially be used in a similar way to check whether the value *null* could flow into a dereference. Our technique, in contrast, is mostly flow- and path-sensitive, using immediate producers only to skip difficult constructs locally. This is in contrast to a full thin slice, which skips a lot more statements (including all conditionals). We presented empirical results in Section 6.5 that show the higher precision of our approach.

There have been earlier approaches that attempt to alleviate the problems due to virtual calls having too many targets, or call-backs having too many predecessors, by pruning edges in the call graph. The Snugglebug approach, which was introduced earlier, performs *directed call graph* construction, wherein they precisely find the targets of a virtual call in a way that is specific to the path that is being currently analyzed. This is orthogonal to our approach to skip difficult virtual calls completely or partially. The approach of Ryder et al. [28], addresses the issue of call-backs by constructing an *application call graph* instead of a whole-program call graph. The application call graph captures (direct or transitive) calling relationships between application methods, eliding library methods entirely. This technique cannot always be used safely in our setting, because the immediate producers of a formula at the entry of a call-back method could potentially be in library code, too.

7.3. Java collections

There exist earlier approaches [29, 30, 31, 12, 32, 13, 15, 33] that share our focus of verifying client code of collections APIs, by abstracting away the implementation details of these APIs. We discuss here these approaches in brief, and compare them with our technique.

The technique proposed by Gregor et al. [29] finds instances of incorrect usage of C++ STL (Standard Template Library) API methods, e.g., attempts to dereference an iterator that had passed the end of the collection it is referring to, and using an out-of-bound index to access elements in `vector`. It uses symbolic execution, but is an unsound bug finding tool. Blanc et al. [32] also propose a technique to verify if client code uses STL correctly. They use predicate-abstraction based model checkers for their analysis. These two approaches do not focus on reasoning about the *contents* of containers, which is the focus of the approaches discussed below as well as of our work.

The approach of Heine et al. [31] uses an ownership-based flow- and context-sensitive type system to detect leaks as well as double-deletes of objects that are stored in polymorphic containers. In this approach the user needs to specify a summary for each API method which indicates both its input-side as well as output-side ownership constraints.

The *Hob* verification framework [14, 30] can be used to verify whether client code that uses data-structure implementations uses them in ways that their consistency properties are maintained. Their approach addresses not just standard (library-provided) data structures, but also user-provided data structures. Client codes that accesses data structures can be analyzed using summary functions of the data structure API methods. The implementations of data-structure methods can also be checked for conformance to the corresponding summary functions. However, the limitation of the approach is that both client code as well as data-structure implementation code needs to be expressed in their own programming language. Moreover, this language does not offer powerful features such as inheritance and virtual calls, which are prevalent in Java programs and which pose challenges to static analysis. This approach has been evaluated only on programs measuring up to about 2000 LOC. However, the kind of properties they address are more deep that just null-ness properties.

The approach for heap analysis of Java programs by Marron et al. [12] uses the semantics of the inbuilt Java collections and iterators methods for analysis of client code, without analyzing the implementations of these methods. They model the pointers in a collection that point to the elements stored in the collection as edges in a heap graph from the node representing the collection object to the nodes representing the elements; our use of the special field *elem* is similar to this. They use a richer notation than us to model the heap accurately as they need to track more properties than us, and not just if a reference may be *null*. However, this potentially hampers the scalability of their approach. They have evaluated their technique only on the Jolden benchmarks (which we have used as micro benchmarks), but not on larger Java applications.

The approach of Dillig et al. [13] models collections precisely, keeping track of positions of elements in *sequence-based* collections such as lists and vectors, which is something we do not do. They also keep track of key-value correlations for maps. They have evaluated their technique on three C++ programs ranging from 16,030 to 128,318 LOC. However, it is not clear whether their approach can be made to work as a backwards analysis, in a demand-driven setting, where efficiency and response time are prime concerns. Moreover, we believe that typical Java programs pose unique challenges that are not predominant in a C++ setting, that impact the scalability of any analysis. As pointed out by earlier work [34], Java programs typically make use of library calls, whose implementations can be large and complex, in a more extensive way than C++ programs. Also, it is generally believed that Java programs are written in a more object-oriented way than C/C++ programs, and are also based on *application frameworks* frequently, which result in an increased prevalence of difficult constructs such as virtual method calls and library call-backs.

Recently, Parízek et al. [15] have proposed an approach for verifying properties of programs that use Java Collections APIs. They are interested in deep properties, e.g., that the keys in a map are all present in some other set, or that a list is maintained in sorted order. They basically model each collection as an array, and express properties using quantified formulas that involve *array theory*. Their approach is based on *predicate abstraction*. They first infer a (finitely bounded) set of predicates for the given program and property, then translate the program into an abstract boolean program, and then model-check this boolean program. The way they construct the abstract boolean program is by defining weakest pre-conditions (WP) rules for the Collections API methods, and then interpreting these rules using an SMT solver. Note that although they specify WP rules, these are not used as dataflow transfer functions, unlike in our setting. Also, since we don't use array theory, and rather use the special fields *elem* and *collection*, our WP rules are quite different from theirs in flavor. Their pre-pass, wherein they construct the abstract program is quite expensive; in their experiments, which were only on small programs ($<$ 65 lines of code), the pre-pass took anywhere from 9 seconds to approximately 15,000 seconds, and also produced large abstract programs.

Arzt et al. [33] have developed a tool *Flowdroid* to perform taint analysis on Android applications. Their taint analysis is basically a forward analysis, which is assisted by backward on-demand alias analysis. As part of the forward analysis phase their tool uses predefined taint-propagation rules, which are essentially summary functions wrt the taint analysis problem, for standard collection classes, string buffers and other commonly used data structures provided by the Java standard library.

It is noteworthy that the approaches mentioned above all employ library summaries in the context of a forward analysis. In a forward analysis, one keeps track of what objects are added to what collections, and the properties (e.g., fields that may be null) of these objects. Therefore, in a statement where we retrieve an object from a collection, one can determine the properties of the object that could be retrieved from the collection at that point. In contrast, in our setting, while traversing a path in the backwards direction and encountering a “retrieve” from an iterator or from a collection, we face a key challenge: we do not know the collection that the iterator is referring to, and we do not know what objects have been added to what collections. Our backwards transfer functions adopt a novel approach to address this challenge, and are entirely different from typical forward transfer functions. For example, in the example in Figure 14, after processing Statements 7 and 6, our formula $\langle \texttt{z}.\mathit{elem} = \mathit{null} \rangle$ asserts that *some* element of the collection pointed by to `z` needs to be null. Then, at an “add” statement, such as Statement 5, we transfer this null-ness property from $\texttt{z}.\mathit{elem}$ to the element being added at that statement, namely, `v`, to result in the formula $\langle v = \mathit{null} \rangle$, and then continue proceeding backwards from this point.

8. Conclusions and future work

In this paper we presented a demand-driven null-dereference verification technique for Java programs. The foundational aspect of this technique is a *base* analysis, which is a flow- and context-sensitive backward data-flow analysis using a lattice of formulas, that over-approximates a weakest at-least once pre-condition computation. We then described two major extensions on top of the base analysis to improve the precision of the approach. The first extension is to address certain difficult constructs on which the flow-sensitive technique of the base analysis incurs much running time while still yielding poor precision. The idea behind the extension is to transmit formulas from program points that follow such constructs directly to statements that affect these formulas, in a manner which is sound, efficient and yet more precise than an (expensive) flow-sensitive analysis of the construct. Our second extension is to employ manually constructed summary functions of Java Collections API methods that over-approximate the weakest at-least once pre-condition semantics of these methods. This extension results in improved precision as well as efficiency. All told, our approach gives good results, in being able to verify approximately 91% of dereferences as safe, on average across ten real benchmarks, with average running time of only 427 ms per dereference.

There are several interesting directions for future work. The first would be to compute immediate producers information in a context-sensitive on-demand manner, rather than use pre-computed immediate producers information which could be less precise. Another would be to apply the immediate producers extension at more kinds of difficult constructs than we do currently. A third would be to introduce manually constructed summaries for library methods other than Java Collections methods. Finally, it would be interesting to explore the use of immediate producers as well as backwards library summaries to improve the precision of other kinds of scalable demand-driven backward analyses.

9. Acknowledgments

The authors wish to thank Ravichandhran Madhavan for his help and suggestions, and the anonymous reviewers as well as Uday Khedker for their extensive suggestions for improving the paper. We also wish to thank IBM Research India and Microsoft Research India for partial funding support for this work.

References

[1] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proc. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77, ACM, New York, NY, USA, 1977, pp. 238–252.

[2] S. Chandra, S. J. Fink, M. Sridharan, Snugglebug: a powerful approach to weakest preconditions, in: PLDI '09: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, 2009, pp. 363–374.

[3] M. Sagiv, T. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, in: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '99, ACM, New York, NY, USA, 1999, pp. 105–118.

[4] R. Madhavan, R. Komondoor, Null dereference verification via over-approximated weakest pre-conditions analysis., in: C. V. Lopes, K. Fisher (Eds.), OOPSLA, ACM, 2011, pp. 1033–1052.

[5] M. Sharir, A. Pnueli, Two approaches to interprocedural data flow analysis, in: S. S. Muchnick, N. D. Jones (Eds.), Program Flow Analysis: Theory and Application, Prentice Hall Professional Technical Reference, 1981.

[6] A. Milanova, A. Rountev, B. G. Ryder, Parameterized object sensitivity for points-to analysis for java, ACM Trans. Softw. Eng. Methodol. (TOSEM) 14 (2005) 1–41.

[7] C. Lattner, A. Lenharth, V. Adve, Making context-sensitive points-to analysis with heap cloning practical for the real world, in: Proc. ACM SIGPLAN Conf. on Prog. Language Design and Implementation (PLDI), pp. 278–289.

[8] S. Muchnick, Advanced compiler design and implementation, Morgan Kaufmann, 1997.

[9] E. Geay, M. Pistoia, T. Tateishi, B. G. Ryder, J. Dolby, Modular string-sensitive permission analysis with demand-driven precision, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 177–187.

[10] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, O. Weisman, Taj: effective taint analysis of web applications, in: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI '09, ACM, New York, NY, USA, 2009, pp. 87–97.

[11] C. Hammer, R. Schaade, G. Snelting, Static path conditions for java, in: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security, PLAS '08, ACM, New York, NY, USA, 2008, pp. 57–66.

[12] M. Marron, D. Stefanovic, M. Hermenegildo, D. Kapur, Heap analysis in the presence of collection libraries, in: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE '07, ACM, New York, NY, USA, 2007, pp. 31–36.

[13] I. Dillig, T. Dillig, A. Aiken, Precise reasoning for programs using containers, in: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11, ACM, New York, NY, USA, 2011, pp. 187–200.

[14] P. Lam, V. Kuncak, M. Rinard, Hob: A tool for verifying datastructure-consistency, in: R. Bodik (Ed.), Compiler Construction, volume 3443 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2005, pp. 137–138.

[15] P. Parízek, O. Lhoták, Predicate abstraction of java programs with collections, in: OOPSLA '12: Proc. ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications, pp. 75–94.

[16] M. G. Nanda, S. Sinha, Accurate interprocedural null-dereference analysis for java, in: ICSE '09: Proc. International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2009, pp. 133–143.

[17] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, M. Nanda, Verifying dereference safety via expanding-scope analysis, in: ISSTA '08: Proc. International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, 2008, pp. 213–224.

[18] F. Spoto, Precise null-pointer analysis, Software and Systems Modeling 10 (2011) 219–252. 10.1007/s10270-009-0132-5.

[19] M. Sridharan, S. J. Fink, R. Bodik, Thin slicing, in: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07, ACM, New York, NY, USA, 2007, pp. 112–122.

[20] Wala, T. J. WAtson Libraries for Analysis, http://wala.sf.net, 2014.

[21] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The dacapo benchmarks: Java benchmarking development and analysis, in: Proc. 21st Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, pp. 169–190.

[22] N. Sinha, N. Singhania, S. Chandra, M. Sridharan, Alternate and learn: Finding witnesses without looking all over., in: P. Madhusudan, S. A. Seshia (Eds.), CAV, volume 7358 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 599–615.

[23] S. Blackshear, B.-Y. E. Chang, M. Sridharan, Thresher: precise refutations for heap reachability, in: PLDI 2013, pp. 275–286.

[24] T. Lev-Ami, T. R. M. Sagiv, S. Gulwani, Backward analysis for inferring quantified preconditions, Technical Report TR-2007-12-01, Tel Aviv University, 2007.

[25] A. Podelski, A. Rybalchenko, T. Wies, Heap assumptions on demand, in: Proc. International Conference on Computer Aided Verification, CAV '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 314–327.

[26] B. Gulavani, S. Chakraborty, G. Ramalingam, A. Nori, Bottom-up shape analysis, in: J. Palsberg, Z. Su (Eds.), Static Analysis, volume 5673 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2009, pp. 188–204. 10.1007/978-3-642-03237-0_14.

[27] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, Scalable shape analysis for systems code, in: Proceedings of the 20th international conference on Computer Aided Verification, CAV '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 385–398.

[28] W. Zhang, B. Ryder, Constructing accurate application call graphs for java to model library callbacks, in: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, SCAM '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 63–74.

[29] D. Gregor, S. Schupp, Stllint: lifting static checking from languages to libraries, Software: Practice and Experience 36 (2006) 225–254.

[30] V. Kuncak, P. Lam, K. Zee, M. C. Rinard, Modular pluggable analyses for data structure consistency, IEEE Transactions on Software Engineering 32 (2006) 988–1005.

[31] D. L. Heine, M. S. Lam, Static detection of leaks in polymorphic containers, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 252–261.

[32] N. Blanc, A. Groce, D. Kroening, Verifying c++ with stl containers via predicate abstraction, in: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ASE '07, ACM, New York, NY, USA, 2007, pp. 521–524.

[33] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. l. Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, in: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation.

[34] L. Hendren, Scaling java points-to analysis using spark, in: Compiler Construction, 12th International Conference, volume 2622 of LNCS, Springer, 2003, pp. 153–169.

[35] P. Sotin, B. Jeannet, X. Rival, Concrete memory models for shape analysis, Electron. Notes Theor. Comput. Sci. 267 (2010) 139–150.

Appendix A. Formalizing the base analysis as an abstract interpretation

In this section we give a formulation of the weakest at-least once precondition analysis of the base analysis as an abstract interpretation [1].

Appendix A.1. Concrete Semantics

$\eta \in C = 2^{State}$	: Concrete Domain
$\sigma \in State = Env \times Store$	: program state
$e \in Env = Var \rightarrow Loc$	: environment
$s \in Store = Addr \rightarrow (Ptr \mid Scalar)$	: store
$Addr = Loc \times (Offset \vert \epsilon)$	: addresses
$l \in Loc$	: Location
$Ptr = Loc \mid null$	
$o \in Offset = FieldName \mid \mathbb{N}$	: offset
$C_1 \sqcup C_2 = C_1 \cup C_2$, where $C_1, C_2 \in C$	: Ordering

Figure A.27: Concrete Semantics

We first introduce in Figure A.27 the concrete lattice that we use to specify the concrete semantics of the statements in the language. Our concrete lattice is inspired by the *standard* memory model described by Sotin et al. [35]. An environment ($e \in Env$) is a mapping from the program variables (Var) to the memory locations (Loc) where their content is stored. A store ($s \in Store$) is a mapping from addresses ($Addr = Loc \times (Offset \cup \{\epsilon\})$) to values $Ptr \mid Scalar$. The scalar values ($Scalar$) are the primitive values. Each address is a pair consisting of a base *Location*, and an *Offset* or 'ϵ'. Offsets are used while referring to constituents of non-primitive memory cells (i.e., objects), while an

'ϵ' is used while referring to a scalar cell. An offset is a declared program field (*FieldName*). As per Figure A.27 offsets can also be natural numbers; however, this option is to be ignored in this section.

Let (e, s) be a state and let v be a variable. The pointer value or scalar value that resides in the location referred to by the access path $v.f_1.f_2 \ldots f_n$ is retrieved by the following recursively defined macro:

$GetContents(e, s, v, f_1, f_2, \ldots, f_n) =$
$\quad (n == 0)\ ?\ s(e(v), \epsilon) : s(GetContents(e, s, v, f_1, f_2, \ldots, f_{n-1}), f_n).$

A concrete state (e, s) is said to satisfy a disjunct ϕ, written as $(e, s) \vdash \phi$, if the disjunct becomes *true* when each access path x in ϕ is substituted with a concrete value from the domain $(Ptr \mid Scalar)$ using the following substitution rules:

1. If x is a variable that belongs to domain *Var*, substitute x with $GetContents(e, s, v)$,
2. else if x is of the form $v.f_1.f_2 \ldots f_n$, substitute x with $GetContents(e, s, v, f_1, f_2, \ldots, f_n)$.

The backward concrete transfer function f_c^b of any statement *st* is as follows: If *st* is not an ASSUME statement then $f_c^b = \lambda\eta.\{(e, s_1) \mid \exists(e, s_2) \in \eta : st \text{ transforms } s_1 \text{ to } s_2\}$. If *st* is of the form "ASSUME b" then $f_c^b = \lambda\eta.\{(e, s) \mid (e, s) \in \eta \text{ and } (e, s) \vdash b\}$.

Let p be the program point that precedes the root dereference, and let C be formula at p reflecting the null-ness hypothesis of the root dereference. The initialization of the *concrete analysis* is as follows: the set of states satisfying C at p, and the empty set at all other program points. It is straightforward to show that the join over all paths solution at the entry of the program according to the above concrete analysis is precisely the set of states that satisfy $wp_1(p, C)$.

Appendix A.2. Abstract semantics

The abstract lattice of the base analysis was presented in Figure 2. Its join operation is set-union (recall that formulas are represented as sets of disjuncts). The *concretization function* γ is $\lambda\phi \in Disjunct.\{(e, s) \mid (e, s) \vdash \phi\}$.

For each statement type *st* the abstract backward transfer function f_d for *st* is shown in Figure 3. It can be shown that each of these abstract transfer functions is monotonic, and also conservatively *over-approximates* the corresponding concrete transfer function f_c^b; i.e., for any disjunct ϕ, $\gamma(f_d(\phi)) \supseteq f_c^b(\gamma(\phi))$. We now prove the above mentioned over-approximation property for the transfer function of the PUTFIELD instruction "$r.f = v$", which is the most involved one of all the transfer functions in Figure 3. We start from the right-side of the inequality above. Consider any disjunct ϕ, and any state (e, s) such that $(e, s) \in \gamma(\phi))$ (i.e., $(e, s) \vdash \phi$). Say ϕ_1 is a disjunct obtained from ϕ as described in Figure A.28. Let (e, s') be any state such that when the statement "$r.f = v$" is executed on this state the state (e, s) results. That is, $(e, s') \in f_c^b(\gamma(\phi))$, where

```
$\phi_1 = \phi$
**for all** access paths $w.f_1.f_2...f_n.f$ in $SubAPs(\phi)$, $n \geq 0$ **do**
  **if** $GetContents(e, s, w, f_1, f_2, \ldots, f_n) = GetContents(e, s, r)$ **then**
    Replace $w.f_1.f_2...f_n.f$ in $\phi_1$ with $v$
    If $MustAlias(w.f_1.f_2...f_n, r)$ is *false* at the point after the statement
    "$r.f = v$" then add the predicate "$w.f_1.f_2...f_n = r$" to $\phi_1$
  **else**
    If $MayAlias(w.f_1.f_2...f_n, r)$ is *true* at the point after the statement
    "$r.f = v$" add the predicate "$w.f_1.f_2...f_n \neq r$" to $\phi_1$
Add the predicate "$r \neq null$" to $\phi_1$
```

Figure A.28: Generating ϕ_1 from ϕ

f_c^b is the backward concrete transfer function of the above instruction. Clearly, for this to happen (without a null-dereference exception), $GetContents(e, s', r)$ cannot be equal to *null*. Given this, and given that $(e, s) \vdash \phi$, it is easy to show that $(e, s') \vdash \phi_1$. It is also easy to show that ϕ_1 is one of the disjuncts returned by $f_d(\phi)$, where f_d is the backward abstract transfer function of the above instruction as described in Figure 3. Therefore, $(e, s') \in \gamma(f_d(\phi))$. Therefore, we have the result that $\gamma(f_d(\phi)) \supseteq f_c^b(\gamma(\phi))$.

Proving the over-approximation property for all other transfer functions in Figure 3 is straightforward.

The initialization for the abstract analysis is as follows: the formula C reflecting the null-ness hypothesis of the root dereference at the point p that precedes the root dereference, and the empty set (i.e., *false*) at all other points. Now, putting all the arguments presented in this section together, and invoking the safety guarantees of abstract interpretation in general [1], it follows that the γ-image of pre-condition computed at the program's entry by the base analysis is equal to or weaker than $wp_1(p, C)$.

Appendix B. Soundness of our Collections transfer functions

In this section we will consider a few representative abstract transfer functions for the Java Collections API from Figure 13 and prove them to be correct abstractions of their respective concrete semantics. Recall that these transfer functions are based on the abstract lattice that was first defined in Figure 2, and then extended in Figure 12.

Appendix B.1. Concrete Lattice

We had earlier introduced the lattice of concrete states (i.e, memory configurations) in Figure A.27. One aspect that we had not discussed in Appendix A was collection objects. Rather than model these objects fully concretely (i.e., as they are implemented), we choose to model collection objects (of all collection types) in a simplified manner as arrays. Recall that an address is a pair consisting of a base *Location*, and an *Offset* or 'ϵ'. When a base location is that of a collection object, we use natural numbers rather than field names as offsets within this object (see the production of *Offset* in Figure A.27). The offset 0 refers to the first element in the collection, 1 refers to the second element, and so on. Although the elements of a collection are indexed by these natural numbers, the abstract transfer functions ignore this ordering. That is, they treat all collections as unordered. An iterator object is modeled as containing a field c, which contains the *Location* of the underlying collection, and a field *next* that contains the offset (a natural number) of the element of the underlying collection that is to be retrieved next.

Recall from the discussion in Section 4 that in order to accommodate references to collections and iterators we use the special fields *elem* and *collection* in our formulas. Therefore, we extend our original definition of a state (e, s) satisfying a formula ϕ (originally given in Appendix A) as follows. $(e, s) \vdash \phi$ holds if the formula ϕ becomes *true* when each access path x in ϕ is substituted with a concrete value from the domain $(\mathit{Ptr} \mid \mathit{Scalar})$ using the following substitution rules:

1. If x is a variable that belongs to domain Var, substitute x with $s(e(v), \epsilon)$,
2. else if x is of the form $v.\mathit{elem}$, substitute x with $s(s(e(v), \epsilon), o)$, where o is some natural number,
3. else if x is of the form $v.\mathit{collection}$, substitute $v.\mathit{collection}$ with $s(s(e(v), \epsilon), c)$, where c is the field in the iterator object referred to by v that refers to the underlying collection,
4. else if x is of the form $v.f_1.f_2 \ldots f_n$, substitute x with $\mathit{GetContents}(e, s, v, f_1, f_2, \ldots, f_n)$.

Appendix B.2. Proofs

We will now prove the correctness of certain representative abstract backwards transfer functions from Figure 13. For each abstract backwards transfer function that we consider in this subsection we show (a) the transfer function f_d itself, from Figure 13, (b) the concrete forward transfer function f_c^f (for ease of understanding), (c) the concrete backwards transfer function f_c^b, and finally (d) the proof that f_d sounds abstracts f_c^b. Note, as discussed in Section 4.3, that our concrete transfer functions capture the concrete semantics of the corresponding API methods in an *idealized* manner. In particular, they assume the simplified array-based representation of collection objects introduced above. In the proofs we use the notation $s[a \mapsto p]$ to represent a store that differs from store s only in that address a contains the value p.

(a) $v = i.\mathit{next}()$
$f_d = \lambda\phi.\, \phi'$, where $\phi' = \phi[i.\mathit{collection}.\mathit{elem}/v]$.
The function *i.next()* returns the next element of the underlying collection object to be iterated. For simplicity, we only consider the case where there is an element to return. Our abstract transfer function is still sound in the scenario

where there is no element to return. In this scenario the concrete semantics is to throw an exception; therefore, the weakest pre-condition of any formula is *false*, which is over-approximated by any abstract state. We will use three utility functions to define the concrete transfer functions: *nextAddr*, *hasNext* and *incNext*. Given a state (e, s) and a variable i that refers to an iterator, the function *nextAddr* returns the address of the "next" element of the collection referred to by i:
nextAddr(e,s,i) $= (\mathit{GetContents}(e, s, i, c), \mathit{GetContents}(e, s, i, \mathit{next}))$
Function *hasNext* takes a state (e, s) and an iterator variable i and returns *true* if the underlying collection referred to by i has a "next" element to be iterated over:
hasNext(e,s,i) = (the address *nextAddr(e,s,i)* is in the domain of s) ? *true* : *false*
Function *incNext* takes a state (e, s) and an iterator variable i and returns an updated store wherein the *next* field of i is incremented:
incNext(e,s,i) $= s[(s(e(i), \epsilon), \mathit{next}) \mapsto (\mathit{GetContents}(e, s, i, \mathit{next}) + 1)]$
Forward concrete function:
$f_c{}^f = \lambda\eta.\, \{(e, s_2) \mid (e, s) \in \eta \wedge \mathit{hasNext}(e, s, i) \wedge s_1 = s[(e(v), \epsilon) \mapsto s(\mathit{nextAddr}(e, s, i))] \wedge s_2 = \mathit{incNext}(e, s_1, i)\}$
Backward concrete function:
$f_c^b = \lambda\eta.\, \{(e, s) \mid (e, s') \in \eta \wedge \mathit{hasNext}(e, s, i) \wedge s_1 = s[(e(v), \epsilon) \mapsto s(\mathit{nextAddr}(e, s, i))] \wedge s' = \mathit{incNext}(e, s_1, i)\}$

To prove : $f_c^b(\gamma(\phi)) \subseteq \gamma(f_d(\phi))$
$f_c^b(\gamma(\phi)) = f_c^b(\{(e, s) \mid (e, s) \in \mathit{State} \wedge (e, s) \vdash \phi\})$
$= \{(e, s) \mid \mathit{hasNext}(e, s, i) \wedge (e, s') \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto s(\mathit{nextAddr}(e, s, i))] \wedge s' = \mathit{incNext}(e, s_1, i)\}$
$\subseteq \{(e, s) \mid (e, s') \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto s(\mathit{nextAddr}(e, s, i))] \wedge s' = \mathit{incNext}(e, s_1, i)\}$
$\subseteq \{(e, s) \mid \exists j \in \mathbb{N}.\, ((e, s') \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto \mathit{GetContents}(e, s, i, c, j)] \wedge s' = \mathit{incNext}(e, s_1, i))\}$
Now, since ϕ does not refer to the *next* fields of iterators, for any stores s' and s_1 such that $s' = \mathit{incNext}(e, s_1, i)$, and for any formula ϕ, $(e, s') \vdash \phi$ iff $(e, s_1) \vdash \phi$. Therefore, we get
$f_c^b(\gamma(\phi)) \subseteq \{(e, s) \mid \exists j \in \mathbb{N}.\, ((e, s_1) \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto \mathit{GetContents}(e, s, i, c, j)])\}$
Now, from substitution Rules 2 and 3 mentioned in Appendix A.1, it follows that $\exists j \in \mathbb{N}.\, ((e, s_1) \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto \mathit{GetContents}(e, s, i, c, j)])$ iff $(e, s) \vdash \phi[i.\mathit{collection}.\mathit{elem}/v]$. Therefore, we get
$f_c^b(\gamma(\phi)) \subseteq \{(e, s) \mid (e, s) \vdash \phi'\}$
Hence, it follows from the definition of f_d that $f_c^b(\gamma(\phi)) \subseteq \gamma(f_d(\phi))$

(b) *c.add*(v)
$f_d = \lambda\phi.\, T$, where T is a set of disjuncts computed as per the pseudo-code shown for the CollectionAdd rule in Figure 13.

c refers to a collection, that is a either a *set* or a *list* (as mentioned in Section 4 every collection in Java is either a set or a list). Initially let us consider the case where c refers to a set.

We first define a few utility predicates to use in the definitions of the concrete functions.

$\mathit{CanbeAdded}(e, s, c, v)$ checks if the object referred to by variable v in state (e, s) can be added to the collection referred to by the variable c in the state (e, s). The logic employed in this predicate is dependent on the kind of collection object referred to by c. For instance, if c refers to a normal set then the predicate would need to check that the object referred to by v is not already present in this set. If c refers to a multi-set then this predicate always return *true*, and so on.

$\mathit{added}(e, s, s', c, v)$ checks if (e, s') is the state obtained by adding the object referred to by variable v in state (e, s) to *some* offset in the collection referred to by variable c in the state (e, s), while preserving all other portions of the store. It is formally defined as:

$\exists i.$ ((the address $(s(e(c), \epsilon), i)$ is unmapped in (e, s)) $\wedge$ $(s' = s[(s(e(c), \epsilon), i) \mapsto s(e(v), \epsilon)]))$

The forward concrete function can be defined as follows:

$f_c{}^f = \lambda\eta.\, \{(e, s') \mid ((e, s) \in \eta \wedge \mathit{CanbeAdded}(e, s, c, v) \wedge \mathit{added}(e, s, s', c, v)) \vee$
$((e, s') \in \eta \wedge !\mathit{CanbeAdded}(e, s', c, v))\ \}$

The transfer function above can be thought of as choosing a free offset within the collection object referred to by the variable c in the state (e, s) non-deterministically.

The backward concrete function can be defined as follows:

$f_c^b = \lambda\eta.\, \{(e, s) \mid ((e, s') \in \eta \wedge \mathit{CanbeAdded}(e, s, c, v) \wedge \mathit{added}(e, s, s', c, v)) \vee$
$((e, s) \in \eta \wedge !\mathit{CanbeAdded}(e, s, c, v))\ \}$

To prove : $f_c^b(\gamma(\phi)) \subseteq \gamma(f_d(\phi))$

$f_c^b(\gamma(\phi)) = f_c^b(\{(e, s) \mid (e, s) \vdash \phi\})$
$= \{(e, s) \mid ((e, s') \vdash \phi \wedge \mathit{CanbeAdded}(e, s, c, v) \wedge \mathit{added}(e, s, s', c, v)) \vee$
$((e, s) \vdash \phi \wedge !\mathit{CanbeAdded}(e, s, c, v))\ \}$
$\subseteq \{(e, s) \mid ((e, s') \vdash \phi \wedge \mathit{added}(e, s, s', c, v)) \vee (e, s) \vdash \phi\}$, by removing some conjuncted conditions. – *(i)*

We now define a utility function $\mathit{Transform}(e, s, \phi, c, v)$, whose pre-requisites are that ϕ is a disjunct that is satisfied by (e, s), c is a variable that refers to a collection in (e, s), and v is a variable that refers to any object in (e, s). This function returns a new disjunct ϕ^t by rewriting ϕ as follows. For each pair of the form $(ap_i.\mathit{elem}, j)$ in $\mathit{SubAPs}^*(\phi)$ (see Section 4.3 for the definition of $\mathit{SubAPs}^*(\phi)$) such that ap_i and c refer to the same (collection) object in (e, s) and such that this collection contains the object referred to by v in (e, s) as one of its elements, replace the jth occurrence $ap_i.\mathit{elem}$ in ϕ by v iff the resulting formula is satisfied by (e, s).

For instance, consider an invocation $\mathit{Transform}(e, s, \phi, c, v)$ wherein ϕ is $\langle k.\mathit{elem} = \mathit{null}, l.\mathit{elem} = x\rangle$, and wherein the parameters satisfy all the pre-requisites of *Transform*. If c and l refer to the same (collection) object in

(e, s), and k does not prefer to this same object, and v and x refer to the same object in (e, s), and one of the elements of the collection referred to by l in (e, s) is this object, then the return value from this invocation is the disjunct $\langle k.elem = null, v = x\rangle$. On the other hand, for instance, if v and x did not refer to the same object in (e, s), then the invocation above would return the original disjunct $\langle k.elem = null, l.elem = x\rangle$ itself.

Lemma 1: Say (e, s) is a state, c is a variable that refers to a collection object in (e, s), v is a variable that refers to some object in (e, s), and (e, s') is another state such that $added(e, s, s', c, v)$ is true. Then, for any formula ϕ, the following holds:
$((e, s') \vdash \phi) \Rightarrow ((e, s') \vdash Transform(e, s', \phi, c, v)) \Rightarrow$
$((e, s) \vdash Transform(e, s', \phi, c, v))$.
Proof:
From the definition of the function *Transform* it is easy to see that $((e, s') \vdash \phi) \Rightarrow ((e, s') \vdash Transform(e, s', \phi, c, v))$. We argue the second implication in the lemma's statement by contradiction, by first assuming that $(e, s') \vdash Transform(e, s', \phi, c, v)$. Say there is some predicate p in $Transform(e, s', \phi, c, v)$ such that (e, s) does not satisfy this predicate. Since this predicate is satisfied by (e, s'), and since (e, s) and (e, s') differ only in that some offset i in the collection referred to by variable c in (e, s) is not mapped to any object in (e, s) while is mapped in (e, s') to the object referred to by v in (e, s) (this is because $added(e, s, s', c, v)$ is true), it follows that (a) p is satisfied by (e, s') due to the element at offset i in the collection referred to by variable c in (e, s), and that (b) p involves an access path $ap_i.elem$ such that ap_i refers to the same collection object as c does in (e, s). It follows from the above observations, and from the definition of *Transform*, that the access path $ap_i.elem$ in p must have been rewritten to simply v by *Transform*. However, this contradicts our earlier finding that p involves the access path $ap_i.elem$. Therefore, we have shown that $(e, s') \vdash Transform(e, s', \phi, c, v)$ implies $(e, s) \vdash Transform(e, s', \phi, c, v)$. □

Now, applying Lemma 1, we can rewrite (i) as
$f^b_c(\gamma(\phi)) \subseteq \{(e, s) \mid (\underline{(e, s) \vdash Transform(e, s', \phi, c, v)} \wedge added(e, s, s', c, v)) \vee (e, s) \vdash \phi\}$

where the underlined portion is the one that has changed relative to (i). Given that $\phi' = f_d(\phi))$, from the definition of f_d in the CollectionAdd rule in Figure 4.3, and by the definition of $Transform(e, s', \phi, c, v)$, it can be shown that $Transform(e, s', \phi, c, v)$ implies one of the disjuncts in ϕ'. Therefore, we get
$f^b_c(\gamma(\phi)) \subseteq \{(e, s) \mid ((e, s) \vdash \phi' \wedge added(e, s, s', c, v)) \vee (e, s) \vdash \phi\}$.
Now, we drop the conjunct $added(e, s, s', c, v))$. Also, since $\phi \Rightarrow \phi'$ as per the definition of f_d, we replace ϕ in the second disjunct in the formula above with ϕ'. Upon simplification we get
$f^b_c(\gamma(\phi)) \subseteq \{(e, s) \mid ((e, s) \vdash \phi')\ \}$.
Therefore, by definition of $\gamma(f_d(\phi))$, we get
$f^b_c(\gamma(\phi)) \subseteq \gamma(f_d(\phi))$.

The same proof holds if v refers to a *list* object rather than a *set*, except that the condition *CanbeAdded* need not be used.

(c) $i = v.iterator()$
$f_d = \lambda\phi.\, \phi'$, where $\phi' = \phi[v/i.collection]$
Forward concrete function:
$f_c^f = \lambda\eta.\, \{(e, s') \mid (e, s) \in \eta \wedge s' = (s[(il, c) \mapsto s(e(v), \epsilon), (il, next) \mapsto 0])\}$, where $il = s(e(i), \epsilon)$. Basically, (il, c) is the c field of the iterator object pointed to by i, while $(il, next)$ is the *next* field of this object.
Backward concrete function:
$f_c^b = \lambda\eta.\, \{(e, s) \mid (e, s) \in State \wedge s' = (s[(il, c) \mapsto s(e(v), \epsilon), (il, next) \mapsto 0]) \wedge (e, s') \in \eta\}$, where $il = s(e(i), \epsilon)$

To prove : $f_c^b(\gamma(\phi)) \subseteq \gamma(f_d(\phi))$
$f_c^b(\gamma(\phi)) = f_c^b(\{(e, s) \mid (e, s) \in State \wedge (e, s) \vdash \phi\})$
$f_c^b(\gamma(\phi)) = \{(e, s) \mid (e, s) \in State \wedge s' = (s[(il, c) \mapsto s(e(v), \epsilon), (il, next) \mapsto 0]) \wedge (e, s') \vdash \phi\}$, where $il = s(e(i), \epsilon)$.

Now, consider any two stores s and s' such that $s' = (s[(il, c) \mapsto s(e(v), \epsilon), (il, next) \mapsto 0])$, where $il = s(e(i), \epsilon)$, and such that $(e, s') \vdash \phi$. Since $s'(e(v), \epsilon) = GetContents(e, s', i, c)$, from Substitution Rule 3 in Appendix A.1 it follows that $(e, s') \vdash \phi[v/i.collection]$. That is, $(e, s') \vdash \phi'$. Now, s and s' differ only in the contents of the addresses $(i1, c)$ and $(i1, next)$. However, since $i.collection$ does not appear in ϕ', and this is the only way to access the location $(i1, c)$, and there is no way in a formula to refer to the *next* field of any iterator, it follows that $(e, s) \vdash \phi'$. Therefore, we get
$f_c^b(\gamma(\phi)) = \{(e, s) \mid (e, s) \vdash \phi'\}$.
Thus, we have shown that $\gamma(f_d(\phi)) = f_c^b(\gamma(\phi))$.

(d) $v.remove(w)$
$f_d = \lambda\phi.\, \phi$
Forward concrete function:
$f_c^f = \lambda\eta.\, \{(e, s) \mid (e, s') \in \eta \wedge IsCollection(v) \wedge (\forall i.\, ((equals(s'(s'(e(v), \epsilon), i), s'(e(w), \epsilon)))$? (the address $(s'(e(v), \epsilon), i)$ is not mapped to any value in s) :
$(matchedEntry(e, s, s', v, i)))))\}$.
We define $matchedEntry(e, s, s', v, i)$ as a predicate that is *true* iff *either* (a) the address $(s'(e(v), \epsilon), i)$ is unmapped in store s' and the address $(s(e(v), \epsilon), i)$ is unmapped in store s, *or* (b) $GetContents(e, s', v, i) = GetContents(e, s, v, i)$.
$IsCollection(var)$ is an auxiliary function to check if *var* refers to one of the library classes that implement `java.util.Collection`.
$equals(x, y)$ is an auxiliary function which takes the locations ($\in Loc$) of two objects and checks if they are semantically equal (using the `equals` method of the object pointed to by x).

Backward concrete function:
$f_c^b = \lambda\eta.\, \{(e, s') \mid (e, s) \in \eta \wedge IsCollection(v) \wedge (\forall i.\, ((equals(s'(s'(e(v), \epsilon), i),$

$s'(e(w),\epsilon)))$? (the address $(s'(e(v),\epsilon),i)$ is not mapped to any value in s) : $(\mathit{matchedEntry}(e,s,s',v,i))))\}$.

To prove : $f_c^b(\gamma(\phi)) \subseteq \gamma(f_d(\phi))$
$f_c^b(\gamma(\phi)) = f_c^b(\{(e,s) \mid (e,s) \in \mathit{State} \wedge (e,s) \vdash \phi\})$
$= \lambda\eta.\,\{(e,s') \mid (e,s) \vdash \phi \wedge \mathit{IsCollection}(v) \wedge$
$(\forall i.\,((\mathit{equals}(s'(s'(e(v),\epsilon),i), s'(e(w),\epsilon)))$? (the address $(s'(e(v),\epsilon),i)$ is not mapped to any value in s) : $(\mathit{matchedEntry}(e,s,s',v,i))))\}$. – *(i)*
Lemma 2: Say (e,s) and (e,s') are *States* that are *identical*, except that for each address *addr* such that a collection object resides in *addr* in s and s', and for each natural number i, *either* (a) the address $(s(e(v),\epsilon),i)$ is unmapped in the store s, *or* (b) $\mathit{GetContents}(e,s',v,i) = \mathit{GetContents}(e,s,v,i)$. Then, for any formula ϕ, $(e,s) \vdash \phi \Rightarrow (e,s') \vdash \phi$.
Proof: Intuitively, since every element of every collection in s is also present in s', and since our formulas *cannot* capture a requirement that a collection *not* contain any element that satisfies a certain property, the lemma follows. $\Box$
Using Lemma 2, continuing from *(i)*, we get
$= \lambda\eta.\,\{(e,s') \mid \underline{(e,s') \vdash \phi} \wedge \mathit{IsCollection}(v) \wedge$
$(\forall i.\,((\mathit{equals}(s'(s'(e(v),\epsilon),i), s'(e(w),\epsilon)))$? (the address $(s'(e(v),\epsilon),i)$ is not mapped to any value in s) : $(\mathit{matchedEntry}(e,s,s',v,i))))\}$ (*the underlined part is the changed part*)
$\subseteq \lambda\eta.\,\{(e,s') \mid (e,s') \vdash \phi\}$ (*by eliminating some conjuncts*)
$\subseteq \gamma(f_d(\phi))$.