Abstract

The problem addressed in this thesis is sound, scalable, demand-driven null-dereference verification for Java programs via over-approximated weakest preconditions analysis. The base version of this analysis having been described in a previous publication, in this thesis we focus primarily on describing two major optimizations that we have incorporated that allow for longer program paths to be traversed more efficiently, hence increasing the precision of the approach. The first optimization is to bypass certain expensive-to-analyze constructs, such as virtual calls with too many possible targets, by directly transferring dataflow facts from points after the construct to points before along def-use edges of a certain kind. The second optimization is to use manually constructed summaries of Java container class methods, rather than analyze the code of these methods directly. We evaluate our approach using 10 real world Java programs, as well as several micro benchmarks. We demonstrate that our optimizations result in a 45% reduction in false positives over the base version on the real programs, without significant impact on running time.

Contents

List of Figures

Chapter 1

Introduction

Null-dereferences are a bane while programming in pointer-based languages such as C and Java. In this thesis, we describe a *sound*, *context-sensitive*, *demand-driven* technique to verify dereferences in Java programs via over-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 $wp_1(p, C)$, 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), $wp_1 p, C = \neg wpp, \neg C$. 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 [3] 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 our over-approximation is satisfiable, we declare the dereference as *potentially* unsafe.

Each element in our data-flow lattice 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

```
1: foo(a,b,c) {
2:   if(a ≠ null) {   ⟨b.f = null, a ≠ null, a = null⟩ → false
3:     b = c;          false
4:     t = new...;     ⟨b.f = null, b ≠ c, a ≠ null⟩
5:     c.f = t;        ⟨b.f = null, b ≠ c, a ≠ null⟩, ⟨t = null, b = c, a ≠ null⟩
6:   }
7:   d=a;              ⟨b.f = null, a ≠ null⟩
8:   if(d ≠ null)     ⟨b.f = null, d ≠ null⟩
9:     b.f.g = 10;     ⟨b.f = null⟩
10:}
```

Figure 1.1: Example to illustrate the base analysis. The dereference in line 9 is being verified.

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 1.1. 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 $\langle \texttt{b.f} = null \rangle$, 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 pre-condition 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 $wp_1(line\ 9, \langle \texttt{b.f} = null \rangle)$, 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 chapters in the thesis.

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 selected set of dereferences 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 only 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 2.4.

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 a formula when it is propagated backwards through a statement that assigns to access paths that are referred to in the formula. A forward analysis can do strong updates by precisely and separately keeping track of aliasing relationships between pairs of access paths along different paths (using *path sensitivity* predicates). This is harder in a backwards analysis; e.g., while analyzing Statement 5 in Figure 1.1, the analysis would have no idea whether this statement definitely updates the value referred to by $b.f$ in the post-condition of this statement. Using pre-computed must-alias information may not be precise enough, because this "all-paths" information is not specific to the individual paths along which the analysis is trying to track distinct formulas (path sensitively).

The second problem is the resolution of *virtual calls* in large object-oriented programs. A backwards analysis would essentially need to depend on a pre-computed call-graph, which itself would be computed using may points-to analysis. Such call graphs often conservatively

over-estimate the set of potential targets of a virtual call, posing a great challenge to a backwards analysis in terms of both scalability and precision. On the other hand, a forward analysis counterpart can potentially use the path-specific and context-specific points-to information to derive a more precise set of targets specific to that path and context.

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* [20] 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 optimizations. The base analysis was originally proposed, discussed, and evaluated by Madhavan and Komondoor [15]. The key elements of the base analysis are the 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 [21] tabulation based approach. Figure 1.1 contains an (intra-procedural) illustration of the base analysis. 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. The base analysis also contains innovative techniques for context-sensitive and path-sensitive analysis which increase precision significantly.

1.3 Our contributions

The primary focus of this paper is the two optimizations in the extended analysis, which address some of the challenges that the base analysis dealt with only in simple ways. The base analysis, when it reaches certain "difficult constructs" that are typically expensive to analyze, yet result in reduced precision, chooses to essentially give up instead of analyzing them. One such scenario is virtual calls with too many potential targets; the base analysis avoids analyzing any of the targets of virtual call-sites that more than a certain number of possible targets (this number being a configurable threshold), and simply reduces all predicates in the post-condition of the call that might be affected by any of the targets (as per a cheap, pre-computed *mod-ref* analysis) to $true$ (thus over-approximating the weakest at-least once pre-condition). Another scenario is when a formula is propagated to the entry of a call-back method, i.e., an application method (such as `equals()`) that is called by library methods; the base analysis analysis chooses to reduce the entire formula to $true$, thereby declaring the dereference in the call-back method as unsafe, rather than pay the price of propagating the formula through the library method.

In the extended analysis we introduce the notion of *immediate producers*, which are a kind of def-use predecessor. In our first optimization we propagate a formula that is a post-condition at a difficult construct directly to statements that are immediate producers of values referred to in the formula, thus skipping all or most of the difficult construct in most cases. Although certain previous approaches [5, 26] have used the *transitive closure* of the immediate-producer relation (which is known as a *thin slice*) to perform certain cheap, approximate analyses, we are the first to our knowledge to use immediate producers within the context of a precise, path-sensitive analysis to skip difficult constructs. Our technique, as such, can be used to skip any kind of difficult construct; however, in our current implementation, we explore the application of our technique to the two kinds of difficult constructs mentioned above.

Since library methods are typically expensive (yet imprecise) to analyze, the base analysis employs several heuristics that it allow to avoid analyzing library calls in various situations, instead approximating their side-effects using simple, conservative shortcuts (e.g., mod-ref analysis). These techniques are library-agnostic, and keep the running time in check, but cause

significant loss in precision. Our second optimization is to target a *specific* category of libraries – the Java Collections API – for special treatment, in a way that is very efficient, yet reasonably precise. Our approach is to use manually-provided transfer functions (i.e., summary functions) for calls to Collections API methods, which we use during the analysis instead of analyzing the implementations of these methods. We target the collections API for such special treatment for two reasons: (a) they are widely used, and contribute much imprecision to the base analysis, and (b) they typically encapsulate a significant percentage of accesses to arrays and recursive data structures, which are typically expensive (and yet imprecise) to analyze. Note that there do exist previous approaches [16, 4, 12, 18] that adopt a similar strategy of verifying of client code of collections by abstracting away the implementation details of the collections; the novelty of our approach 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 optimizations 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 has gone down from 16.25% by the base analysis to 9% by the extended analysis, which is a 45% reduction. We measure the running time of our analysis (as well as the base analysis) by running the analysis separately on each dereference in a program, without any sharing of analysis results between these analyses. The running time of the base analysis analysis is on average 288 mS for a dereference, while for the extended analysis the corresponding number is 427 mS. The running time is higher with the extended analysis because we now abort the analysis of paths in fewer scenarios, and instead continue pursuing longer paths in an effort to prove the root dereference safe. Still, 427 mS is very good even in the context of a demand-driven setting.

The rest of this paper is structured as follows. We give an overview of the base analysis in Chapter 2. In Chapter 3 we present our first optimization, which is based on using immediate producers to skip "difficult" programming constructs. We then present our second optimization, to handle the built-in Java Collections APIs using manually provided summary functions, in Chapter 4. Chapter 5 has a discussion on the results of applying our optimizations on real programs. Chapter 6 contains a discussion of related work. Finally, there is an appendix, with

several parts. Appendix F contains proofs of correctness of some of our key Collections API summary functions. Appendix D discusses some key details of our implementation of our analysis. The other parts of the appendix contain various details of our approach that have been omitted from the main writeup for the sake of brevity.

Chapter 2

Base analysis

The base analysis checks if a 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 [15].

2.1 Abstract lattice and transfer functions

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

Figure 2.1: Structure of formulas (lattice elements) in base analysis

The data-flow lattice of the base analysis is described in Figure 2.1. Each lattice element is a $Formula$, which is a set of $Disjuncts$. Each $Disjunct$ is a set of $Predicates$. Any $f \in Formula$ is a set that can be logically interpreted as the disjunction of its elements; whereas,

any $d \in Disjunct$ 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 $f_1, f_2 \in 2^{Disjunct}$. 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 its containing disjunct. This in general results in an over-approximation, as the resulting formula after dropping a predicate is weaker than the original formula.

We assume that the program is in an Intermediate Representation (IR) form like three-address code. The (backward) transfer functions for the individual statements in the IR are shown in Figure 2.2. These functions are *distributive*; therefore, we express each function 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 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 e.g., this happens in line 5 in Figure 1.1, where `b.f` gets rewritten to `t` in one of the disjuncts. Our convention is to always underline the root predicate.

We now discuss the transfer functions shown in Figure 2.2. The functions COPY, NULLASGN, and RETURN are self-explanatory; we discuss below some of the more interesting

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	$v = r.f$	$\phi[r.f/v] \cup \{r \neq null\}$
ASSUME	$assume(b)$	$\phi \cup \{b\}$, 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
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 $SubAPs(\phi)$ that end with field f, and $S[r.f, v, ap_i.f]$, where S is a set of disjuncts, is $= \{\phi_j[v/ap_i.f] \cup \{r \neq null\} \mid \phi_j \in S\}$, if $MustAlias(r, ap_i)$ after $r.f = v$ $= \{\phi_j[v/ap_i.f] \cup \{r = ap_i, r \neq null\} \mid \phi_j \in S\} \cup$ $\{\phi_j \cup \{r \neq ap_i, r \neq null\} \mid \phi_j \in S\}$, if $MayAlias(r, ap_i)$ after $r.f = v$ $= \{\phi_j \cup \{r \neq null\} \mid \phi_j \in S\}$, otherwise
RETURN	$return\ v$	$\phi[v/ret]$, where ret is a place-holder for the return value

Figure 2.2: Abstract transfer functions used in the base analysis

ones. The EXPRASGN function *drops* all predicates in ϕ that depend on v ($Vars(pred)$ 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). The base analysis implements a limited notion of path sensitivity; 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 in ϕ, the transfer function produces a separate disjunct in the pre-condition 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. Under the second hypothesis the analysis produces a disjunct that is the same as ϕ, with the *alias predicate* $\langle r \neq ap_i \rangle$. 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 Figure 1.1. 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 $SubAPs(\phi)$; 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 e.g., $SubAPs(\langle v.f = null, v = u.g \rangle)$ is $\{v, v.f, u, u.g\}$. Note also that the transfer function makes use of pre-computed *MayAlias* and *MustAlias* information (if available). This is just an optimization for efficiency with no influence on the precision of the base analysis. If there is no pre-computed may-alias information available then one may assume that every access path is may-aliased with every other access path of compatible type, and that no two access paths are must-aliased (unless they are syntactically identical).

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

Figure 2.3: Rules for simplifying disjuncts

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 2.3 shows a sampling of the rules used in the simplifier. Rules 2 and 3 reduce a disjunct to $true$ or $false$; the other rules reduce individual predicates (in disjuncts) to $true$ or $false$. For e.g., the disjunct just above statement 2 in Figure 1.1 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's ideally possible. Thus, the soundness of the analysis is preserved.

The soundness of the transfer functions used in the base analysis has been shown by formulating it as an abstract interpretation. We refer the reader to our earlier paper [15] for the details of this formulation.

2.2 Interprocedural analysis

The inter-procedural aspect of the base analysis is based on Sharir and Pneuli's tabulation based approach [21] for context-sensitive analysis. Whenever a disjunct reaches the point

after a procedure call-site, it is propagated to the end-point of the method(s) that are potential targets of this call-site. When these disjuncts reach the entry-points of these methods they are propagated back to the point that precedes the call-site. When the root dereference is in a method m, or in a callee of m, and the propagated disjuncts reach the entry of m, then there is no specific caller of m to return to. Therefore, these disjuncts are propagated to the point before *each* call-site in the program that calls m; we call the methods that contain these call-sites the *predecessors* of m. In order to minimize space we relegate the discussion of certain other aspects of the inter-procedural propagation to Appendix A.

2.3 Example

We use the example in Figure 1.1 to illustrate the base analysis. The *root dereference*, i.e., the given dereference that we wish to verify, is the dereference of `b.f` at line 9; hence we start the analysis with the singleton disjunct $\langle \texttt{b.f} = null \rangle$ at the point above this dereference, and the empty set at all other points. Predicate $\texttt{d} \neq null$ 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 $\texttt{b.f} = null, \texttt{a} \neq null$ 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.

2.4 Advantages of a backwards analysis

Approaches for null-dereference verification that preceded our publication [15] on the base analysis, e.g., [14, 24], are based on a *forward* analysis, and are meant to verify *all* dereferences in a program in one go. A backwards analysis supports verification of a *single* selected dereference. This is an valuable feature in real world development, where most changes are incremental and affect only a small part of the code base. Thus, we can verify only the part of code that is modified (along with the portion of the program that precedes this), without focusing on the rest of the code. 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 of only necessary aliasing information. For instance, in Figure 1.1, if there were statements between lines 7 and 8 that copied `a` to additional variables other than `d`, since those variables do not occur in the pre-condition before line 8 (shown to the right of line 8), these aliasing relationships (between `a` and the other variables) will not be tracked at all. A forward analysis, in contrast, would need to track *all* aliasing relationships in order to achieve similar precision as the base analysis. (b) Similarly, a backwards analysis need not keep track of null-ness or non-null-ness of *all* access paths that occur in the program; only access paths that occur in the disjuncts that get propagated from the root dereference need to tracked. (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, as soon as any disjunct gets validated (the root dereference is unsafe) or all disjuncts get invalidated (the root dereference is safe). 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. It is this demand-driven nature of the base analysis that enables it 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 chapters of this thesis, entirely preserves this backward and demand-driven style of analysis, while incorporating optimizations to enhance its precision.

Chapter 3

Improving our analysis using producer-consumer edges

There are two *difficult* constructs in real programs, which if analyzed normally without any limits, were observed to cause the running time of the base analysis to become very high. These difficult constructs are: (1) *Difficult virtual calls*, by which we mean virtual calls that resolve to too many potential targets as per a pre-computed may-points-to analysis. Normally, upon encountering a virtual call, the base analysis propagates the post-condition ϕ at the call-site through *all* potential targets of the call. However, if the number of potential targets is greater than a pre-set limit (which was 10 in the experiments), the analysis was found to be come prohibitively expensive, and was hence configured to not enter any of the targets. Rather, as an over-approximation strategy, all predicates in ϕ that might be affected by any of the potential targets (as per a pre-computed *mod-ref* analysis) are simply dropped to yield the pre-condition. (2) *Call-backs*, which are application methods that are called by library code (e.g., user-defined `equals()` methods). When the root dereference is inside a call-back method m, or a callee of a call-back method m, then, as was discussed in Section 2.2, any disjunct that reaches the entry point of m would need to be propagated to all *predecessors* of m, i.e., library methods, and then eventually, to all predecessors of these methods in the client code. This is an expensive thing to do; to avoid it, the base analysis is configured to give up entirely (and call the root dereference unsafe) whenever a disjunct reaches the entry of a call-back method.

```
1:  u = new B();
2:  x = new A();
3:  x.f = u;          ⟨u = null⟩
4:  y = new A();
5:  y.f = v;          ⟨v = null⟩
6:  z = x
7:  if(a < 2)
8:    z = y;
9:  e = z.f;          ⟨z.f = null⟩
10: print e.g
```

Figure 3.1: Example to illustrate immediate producers and other producers. Immediate producers of z.f at the point before Statement 10 are underlined. Other producers of Statement 10 are wavy underlined.

It was observed that if these limits on analysis of difficult virtual calls and call-backs are not enforced then the analysis time on average increased by about *eighty times* over enforcing the limits.

These limits make the base analysis feasible, but cause significant precision loss. On average, over all the real programs analyzed, about 26% of the dereferences in each program that are reported as unsafe by the base analysis are characterized so because of the limits mentioned above. In particular, 13% experienced the virtual-call limit, 10% experienced the call-back limit, while 3% experienced both limits (along different paths).

In this section we describe our first major optimization, based on *immediate producers*, to add back some of the precision lost due to these limits. We found that 68% of the dereferences that were declared unsafe by the base analysis due to the limits are declared as safe as a result of our optimization.

3.1 Immediate producers

We define the notion of *immediate producers* 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 for a purpose other than pointer dereferencing, and (2) q may write to l, and (3) there is a path from q to p along which l may not be written to by any statement. For instance, in the example program in Figure 3.1, Statements 3 and 5 (which are underlined) are the immediate producers of the access path `z.f` at the point before Statement 9. Note that Statements 6 and 8 are *flow-dependence* [10] (or *def-use*) predecessors of `z.f` at Statement 9, but are *not* immediate producers, because the value in `z` is used only for dereferencing in `z.f`. 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. The immediate producers are always a subset of the flow-dependence producers.

Our notion of immediate producer is derived from the notion of *producers*, introduced by Sridharan et al. [25]. 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 for a purpose other than pointer dereferencing. In the example in Figure 3.1 the producers of Statement 10 are the statements that are normally underlined or wavy underlined. Note that if we had defined the immediate producers of a 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. For example, Statement 1 is an immediate producer of Statement 3.

A key observation we make is as follows: 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 may copy the value k. For instance, the predicate shown to the right of Statement 9 can be true only if the predicate shown to the right of Statement 3 or to the right of Statement 5 is true. The correctness of this observation is easy to see: 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 optimization that we discuss in this chapter is based on this observation.

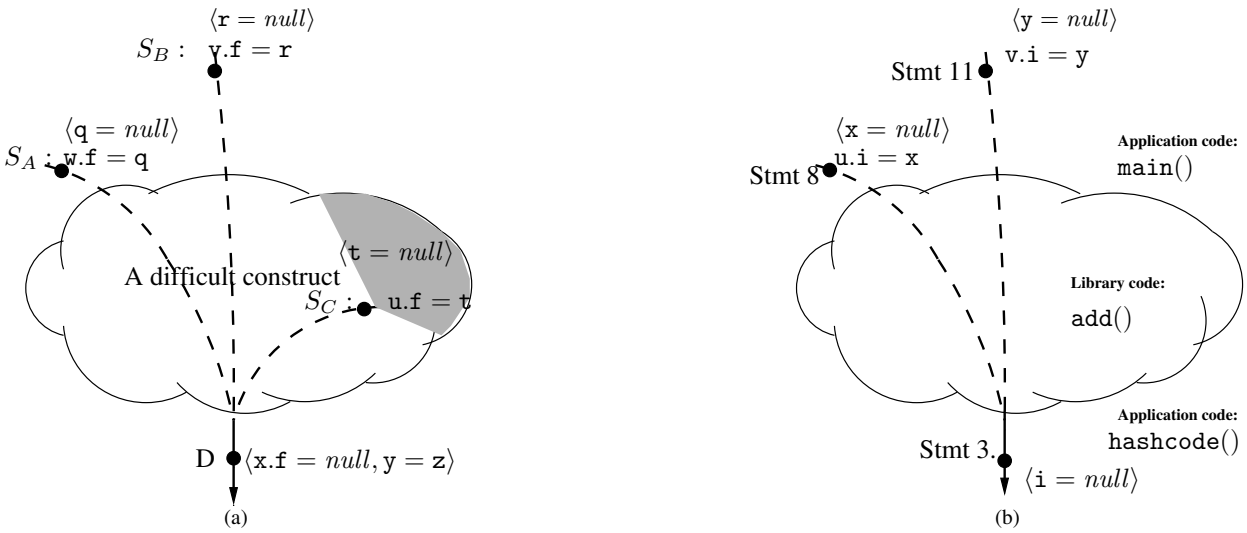


Figure 3.2: (a) Propagation of the disjunct at D to sites producing values for it on encountering *Difficult construct*. (b) Illustration of our approach on the example in Figure 3.6.

Note the "only if" in the observation. For instance, in the example, say the code that precedes the fragment in Figure 3.1 is such that v has value $null$ only when a is greater than or equal to 2. In this case, since the predicate to the right of Statement 5 can be true, we declare the predicate to the right of Statement 9 as being possibly true. However, it can never be true because whenever Statement 5 copies the $null$ value this value will not reach z (due to the condition in line 7). In other words, our observation potentially leads to over-approximating the truth value of the predicate we are reasoning about.

3.2 Skipping difficult constructs using immediate producers

Our optimization, which is basically to *skip* difficult constructs, is a generic one, which can potentially be used to skip *any kind* of difficult construct (in a conservative manner). The idea is simple: Consider a disjunct ϕ at a program point p such that p is preceded in the control-flow graph by a "difficult" construct, which the analysis is trying to avoid having to propagate the disjunct through in the normal manner. Since the observation we made earlier applies only to a *single* predicate involving an access path and a constant, we *drop* from ϕ all predicates except the root predicate "$\langle AP = null \rangle$", and *carry* this simplified disjunct to the points that precede

the immediate producers of *AP* at point p. For instance, if an immediate producer is of the form "*AP1 = AP2*", the carried predicate (at the point that precedes this immediate producer) would be of the form "$\langle AP2 = null \rangle$". We then apply an instance of our *complete* analysis separately on *each* of the carried predicates; i.e., we treat each of the carried disjuncts as if it resulted from a root dereference at that point. Once these carried disjuncts have all been analyzed we reduce ϕ at p to $false$ if the weakest at-least-once pre-conditions of all the carried predicates are found to be *false*, and to *true* otherwise. This will result in the difficult construct being completely skipped in case all the immediate producers are outside this construct, or else in it being partially skipped.

We illustrate the above idea using Figure 3.2(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 $\langle \underline{\texttt{x.f} = null}, \texttt{y} = \texttt{z} \rangle$. "$\texttt{x.f} = null$" 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 "$\texttt{y} = \texttt{z}$", 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. The shaded portion above s_C represents the portion of the difficult construct that still needs to be analyzed (i.e., was not skipped successfully). Note that the reason we drop the predicate "$\texttt{y} = \texttt{z}$" 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. The carried predicates are then checked individually using our complete wp_1 analysis.

The above mentioned approach is described formally in Figures 3.3 and 3.4. Algorithm *transmitDisjunctToImmProd* takes a disjunct ϕ at a program point *p* as the input. It returns a set of tuples of the form (s_i, ϕ_i), where s_i is an immediate producer of the access path in the root predicate of ϕ at p, and ϕ_i is the disjunct obtained by carrying ϕ to the point that precedes s_i. This algorithm uses a few utility methods: *getRootAP(ϕ)*, which returns the root predicate in the disjunct ϕ, *ImmProds(rootAp,p)*, which returns the immediate producers of *rootAp* at point *p* (we discuss how we implement this in Appendix D), and *getRhsAP(s)*, which returns the

Algorithm: *IsInvalidUsingTS*

Input: (p,ϕ), where p is a program point and ϕ is a disjunct at p.

Output: $true$, if ϕ can be invalidated, $false$ otherwise.

Step 1: *transmittedDisjSet* = *transmitDisjunctToImmProd*(p,ϕ)

Step 2: **IF** *transmittedDisjSet* is empty

Step 3: *result* = $false$

Step 4: **ELSE**

Step 5: *result* = $true$

Step 6: **FOR ALL** (s,ϕ') $\in$ *transmittedDisjSet*

Step 7: Apply an instance of our complete analysis
treating ϕ' at the point before s as the initial disjunct.

Step 8: **IF** the above analysis did *not* invalidate ϕ'

Step 9: *result* = $false$

Step 10: **BREAK**

Step 11: **END IF**

Step 12: **END FOR**

Step 13: **END IF**

Step 14: **RETURN** *result*

Figure 3.3: Algorithm *IsInvalidUsingTS* checks if the input disjunct can be invalidated by our approach of using immediate producers.

right-hand side of an assignment statement s.

Algorithm *IsInvalidUsingTS* is the main algorithm. Whenever during propagation of disjuncts in our main analysis a disjunct ϕ reaches a program point p that is preceded by a difficult construct we call this algorithm with arguments p and ϕ. This algorithm first uses *transmitDisjunctToImmProd* to carry ϕ to points that precede the immediate producers of ϕ at p. It then applies our complete analysis on each of the carried disjuncts, and returns $true$ iff each of these disjuncts got invalidated (return value $true$ means ϕ can be invalidated). 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, we discuss in Appendix D.2.

Algorithm: *transmitDisjunctToImmProd*

Input: (p,ϕ), where p is a program point and ϕ 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 ϕ, and ϕ_i is this root predicate carried over to the point that precedes s_i.$\}$

Step 1: *result* = $\emptyset$

Step 2: **IF** ϕ has a *root predicate*

Step 3: *rootAp* = *getRootAP*(ϕ)

Step 4: *prodSet* = *ImmProds*(*rootAp*,*p*)

Step 5: **FOR ALL** statement *s* in *prodSet*

Step 6: *ap* = *getRhsAP*(s)

Step 7: ϕ' = $\langle ap = null \rangle$

Step 8: *result* = *result* $\cup$ $\{$(*s*,ϕ')$\}$

Step 9: **END FOR**

Step 10: **END IF**

Step 11: **RETURN** *result*

Figure 3.4: Algorithm *transmitDisjunctToImmProd* transmits the input disjunct to the immediate producers of the root predicate in the disjunct.

Carrying a predicate to its immediate producers potentially causes an over-approximation (as noted at the end of Section 3.1); furthermore, dropping predicates also may cause over-approximation. Therefore, it is easy to see that the weakest at-least once pre-condition of ϕ at p is over-approximated by the disjunction of the weakest at-least once preconditions of the carried disjuncts at their respective points.

3.3 Instantiation of our approach

We now discuss how we instantiate the generic idea that we described above to two particular kinds of difficult constructs – difficult virtual calls, and library call backs. Whenever in the extended analysis a disjunct ϕ reaches a point p that is after a *difficult virtual call*, as was defined earlier in this chapter, and one or more targets of this call have the potential to affect

```
1:  foo(u,v) {
2:    u.f = v;
3:  }
4:  main() {
5:    b = new..();
6:    b.g = new..();    ⟨ t_6 = null ⟩
7:    a = new..();
8:    t.foo(a,b)
9:    ...               ⟨a.f.g = null⟩
10: }
```

Figure 3.5: Example to illustrate that even a method with side-effects can be skipped using immediate producers.

ϕ as per the pre-computed mod-ref information, we call the procedure *IsInvalidUsingTS*(p, ϕ). We then reduce ϕ to $false$ if *IsInvalidUsingTS* returns $true$, else we fall-back to what the base analysis would do, as was described earlier.

Consider, for e.g., the fragment in Figure 3.5. Say the call in line 8 resolves to too many potential targets, with the method in lines 1–3 being one of the targets. The post-condition after the call involves the access path a.f.g, as shown. Even though line 2 can affect the post-condition, the immediate producer of a.f.g in the post-condition is *not* Statement 2; rather, it is Statement 6, which lies outside the method foo(). Our approach would directly carry the post-condition after Statement 8 to the point before Statement 6; there, it would become $t_6 = null$, where t_6 is the special variable representing objects allocated in Statement 6. This disjunct gets invalidated immediately by the simplification rules shown in Figure 2.3.

Similarly, whenever a disjunct ϕ reaches the entry point p of any library call-back method, in the extended analysis we apply the procedure *IsInvalidUsingTS*(p, ϕ). We reduce ϕ to $false$ (thus calling the root-dereference safe) if the procedure returns $true$, else we reduce ϕ to $true$ (thus calling the root-dereference unsafe).

Code	Extended analysis
1: public class Element {	
2: Integer i;	
3: public int hashCode() {	**Entry of Call-Back hit !!!**
4: return **i.hashCode()**;	$\langle \underline{\texttt{i} = null} \rangle$
}	
}	
main() {	
5: z= new HashSet();	
6: Element u = new Element();	
7: Integer x = new Integer(0);	$false$
8: u.i = x;	$\langle \underline{\texttt{x} = null} \rangle$ (*immediate producer of* `i` *at line 3*)
9: Element v = new Element();	
10: Integer y = new Integer(1);	$false$
11: v.i = y;	$\langle \underline{\texttt{y} = null} \rangle$ (*immediate producer of* `i` *at line 3*)
12: z.add(u);	
13: z.add(v);	
}	

Figure 3.6: Example to illustrate verification of a dereference (indicated in bold) inside a call-back method.

Consider the example in Figure 3.6. 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), we carry it to the points that precede its immediate producers, which are Statements 8 and 11. The result of this carrying is shown in Figure 3.6, as well as in Figure 3.2(b). (In fact, in this example, we would have had the exact same outcome had we propagated the disjunct at the entry of

```
1:   z = new A();
2:   y = new B();
3:   if(z == null)
4:     y = null;
5:   z.f = y;
6:   v = z.f;
7:   print v.g;
```

Figure 3.7: Thin slice from dereference of `v` at Statement 7

the method `Element.hashCode` via the code of the library method `HashSet.add` 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.

Note that the reason that the base analysis analysis is designed to give up right at the entry of library call-back methods is that library code can be vast; also, the analysis is likely to end up losing all precision anyway within library code due to the use of complex data structures such as arrays, recursive data structures, etc. Note also that our approach is very effective in carrying disjuncts from library call-back methods straight to the application code, completely bypassing the library code, while being applicable across *all* library methods; in fact, our experiments using real programs 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).

We provide some additional details about our approach to handling virtual calls and library calls in Appendix B.

3.4 Novelty of our approach

Certain previous approaches use a full *thin-slice* [25] of a program, which is the transitive closure of the *producer* relation, to solve *specific* problems. For instance, Tripp et al. [26]

apply this technique to find security vulnerabilities in web applications; Geay et al. [5], for finding the minimum permission for a component to run in a large assembled program; and Hammer et al. [8], to identify potential run-time types of object references in Java programs. We did consider an approach similar to the ones referred to above, to directly use a thin-slice to do null-dereference verification. 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. While this approach is sound and efficient, we found it to be extremely imprecise in our context. For instance, consider the example in Figure 3.7, wherein the thin slice from the dereference of `v` in Statement 7 is shown using underlining. The presence of Statement 4 causes this approach to inaccurately characterize the selected root dereference as unsafe. Note that it would not be possible to make the approach more precise by performing a flow- and path-sensitive analysis *on top* of the thin slice, e.g., because the thin slice is not closed wrt flow-dependences. For instance, Statement 1, on which Statement 7 is transitively dependent, is missing from the thin slice.

Our idea of using immediate producers is much more fine-grained, in the sense that we analyze the full-program flow-sensitively and path-sensitively as far as possible for high-precision, and use immediate producers only locally to bypass certain difficult constructs. In other words, our contribution is to integrate the propagation of dataflow facts along certain kinds of def-use edges under certain situations on top of a base flow-sensitive analysis. Our idea is hence a generalization of the approach used by the previously proposed thin-slice-based analyses. Our approach is also likely to be useful in other contexts that require backwards propagation of formulas.

A related idea is to run our analysis on a (normally) sliced version of the program, rather than on the whole program. However, it was observed [15] that the time saved due to analysis of a smaller program was more than offset by the time spent on computing the slice.

Chapter 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 2.2), 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 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 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 chapter.

4.1 The special fields $elem$ and $collection$

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

Figure 4.1: Our structure of formulas (lattice elements) to handle collections

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 4.1. The rest of the grammar describing the syntax of formulas remains the same as in Figure 2.1. 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 of the collection. Therefore, e.g., 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*.

- $collection$: 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 2.3 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

The Java collections methods are the ones declared in the interface `java.util.Collection`, as well as the ones declared in its subinterfaces `java.util.Set` and `java.util.List`. Our summary functions for the collections methods, which we also call *transfer functions* or *rules*, are shown in Figure 4.3; for brevity we omit some of the simple and obvious transfer functions from the figure.

Before discussing the details of these transfer functions, we first intuitively illustrate our approach using the complete example in Figure 4.2. 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 $\langle \texttt{w} = null \rangle$ 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 $\texttt{i}.collection.elem$, yielding the pre-condition $\langle \texttt{i}.collection.elem = null \rangle$, 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 4.3.

At Statement 6 it gets revealed that `i`'s iterator refers to the collection `z`; therefore, in our predicate, we replace $\texttt{i}.collection$ (the collection referred to by `i`) with `z`, resulting in the condition $\langle \texttt{z}.elem = null \rangle$. This is taken care of by the GETITERATOR transfer function. At Statement 5, we generate two disjuncts in the precondition: (1) $\texttt{v} = null$, reflecting the hypothesis that the element referred to by $\texttt{z}.elem$ in the post-condition is exactly the element pointed to by `v`, which is being added to `z`, (2) $\texttt{z}.elem = null$, 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 $\texttt{z}.elem$, which refers to the elements in the collection, can be $true$ just after this statement. Hence, the transfer function INIT reduces the disjunct $\langle \texttt{z}.elem = null \rangle$ to $false$. The remaining two disjuncts, namely, $\langle \texttt{u} = null \rangle$ and $\langle \texttt{v} = null \rangle$, get invalidated at Statements 1 and 2, respectively. Thus, our analysis proves that the dereference at Statement 8 is safe.

```
 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 4.2: 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.

4.3 Discussion on the transfer functions

In this section we discuss the key details of some of the transfer functions in Figure 4.3. Basically, rules GETITERATOR, ITERATORNEXT, and COLLECTIONADD were illustrated in the example above. Of these, Rule COLLECTIONADD is the most complex one. As mentioned earlier, when processing a statement '$c.add(v)$', we need to hypothesize for each access path $ap_i.elem$ that occurs in the post-condition ϕ, such that ap_i is may-aliased with c at that point, that $ap_i.elem$ may or may not refer to the element pointed to by v. Therefore, we generate a disjunct in the pre-condition corresponding to each of these two hypotheses. In fact, if the same access path $ap_i.elem$ occurs two times in ϕ, since they may not be referring to the same element of the collection, we generate four disjuncts in the pre-condition. In general, if there are k occurrences of sub-access paths of the form $ap.elem$ in ϕ, we generate upto 2^k disjuncts in the pre-condition.

The COLLECTIONADD rule uses a utility function $SubAPs^*(\phi)$, which we introduce now. This function is different from the function $SubAPs(\phi)$ used in Figure 2.2. While ***SubAPs***(ϕ) returns all the possible sub-access paths in ϕ, $SubAPs^*(\phi)$ returns the set

$\{(ap.elem, i) \mid ap.elem \in SubAPs(\phi)$, $1 \leq i \leq n$, n is the number of occurrences of $ap.elem$ in $\phi\}$.

For e.g., if ϕ is $\langle \texttt{x}.elem.\texttt{f} = null, \texttt{x}.elem = \texttt{y}.elem \rangle$ then $SubAPs(\phi)$ is $\{\texttt{x}, \texttt{x}.elem, \texttt{x}.elem.\texttt{f}, \texttt{y}, \texttt{y}.elem\}$, whereas $SubAPs^*(\phi)$ is $\{$ $(\texttt{x}.elem, \texttt{1})$,$(\texttt{x}.elem, \texttt{2})$, $(\texttt{y}.elem, \texttt{1})\}$, where $(\texttt{x}.elem, \texttt{1})$ denotes the 1^{st} occurrence of the sub-access-path $\texttt{x}.elem$ in ϕ, and so on. This utility function is basically used in the transfer function to identify the set of all occurrences of sub-access-paths of the form $\texttt{ap}.elem$ in ϕ; what happens then is that for each *subset* of this set 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*).

Note that in Figure 4.3 we have defined the substitution operator $\phi[c.elem, v, (ap_i.elem, j)]$ as operating on a single disjunct ϕ. However, it actually needs to operate on a *set* of disjuncts (just like the substitution operator in the PUTFIELD rule in Figure 2.2). The extension of this operator to a set of disjuncts is straightforward: we simply apply the operator individually on each disjunct in the set, and union the resulting sets of disjuncts.

Two interesting rules that did not get used in the example in Section 4.2 above are LISTGET and REMOVE. 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, 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. The rule REMOVE is actually used to model three (related) API methods: $v.remove(o)$, $v.removeAll(c)$, and $v.retainAll(c)$. The first of these methods removes element o 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 4.1) depends only on what elements are present in a collection, and not on what elements are *not* present in a collection.

Appendix C contains some additional discussion of Rules LISTGET and COLLECTION-ADD, as well as of certain other rules in Figure 4.3 that were not discussed above. Appendix F contains proofs of correctness of four key rules in Figure 4.3.

4.4 Novelty

There exists earlier work [16, 11, 4, 1, 18] that shares our focus of verifying application code that uses standard collections by abstracting away the implementation details of the collections. In particular, the $elem$ field that we use is similar to special constructs used in some of these earlier approaches [16, 11] to model elements of collections. However, we are the first technique to use manual summary functions of Collections API methods as backwards transfer functions in the context of propagating formulas. The approach of Parízek et al. [18] constructs an *abstract boolean program* first, and then model checks this program (using a forward analysis). The other approaches mentioned above analyze the given program itself by propagating formulas (like us), but in the forward direction. 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 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 the corresponding forward transfer functions. For example, after processing Statements 7 and 6, our formula $\langle \mathtt{z}.elem = null \rangle$ 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 $\texttt{z}.elem$ to the element being added at that statement, namely, v, to result in the formula $\langle v = null \rangle$. The benefits of our backward analysis, namely, that it is demand-driven in many ways, were discussed earlier in Section 2.4.

4.5 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, $\texttt{m}.keys$ (its set of keys), and $\texttt{m}.values$ (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 $\texttt{m.put(k, v)}$ as a sequence of two *add* operations: $\texttt{m}.keys.\texttt{add(k)}$ and $\texttt{m}.values.\texttt{add(v)}$. Note that as per the specified semantics of maps the operation $\texttt{m.get(k)}$ returns null when the key referred to by k is not present in m. Therefore, we translate operation "$\texttt{m.get(k)}$" as "$(...)\ ?\ \texttt{map.values.get(k)} : null$", 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 Section D). The other methods in the `Map` interface are handled in a similar way, conservatively.

	Container operation	**Transfer Function**: $\lambda\phi \in Disjunct.\phi'$, where $\phi' \in 2^{Disjunct}$, and is =
ITERATORNEXT	$v = i.next()$	$\phi[i.collection.elem/v]$
GETITERATOR	$i = v.iterator()$	$\phi[v/i.collection]$
COLLECTIONADD	$c.add(v)$	$\{\phi\}[c.elem, v, (ap_i.elem, q)][c.elem, v, (ap_j.elem, r)] \ldots$ $[c.elem, v, (ap_m.elem, u)]$, where $\{(ap_i.elem, q), \ldots (ap_m.elem, u)\} = SubAPs^*(\phi)$, and $\phi[c.elem, v, (ap_i.elem, j)]$ $= \phi[v/(ap_i.elem, j)] \cup \{c \neq null\}, \phi \cup \{c \neq null\}$, if $MustAlias(c, ap_i)$ $= \phi[v/(ap_i.elem, j)] \cup \{c = ap_i, c \neq null\}$, $\phi \cup \{c \neq ap_i, c \neq null\}$, if $MayAlias(c, ap_i)$ $= \phi \cup \{c \neq null\}$, otherwise
COLLECTIONCLEAR	$v.clear()$	$\phi[false/pred_1][false/pred_2] \ldots [false/pred_n]$, where the $pred_i$'s are the predicates in ϕ that that contain $v.elem$ as a sub access path.
COLLECTIONADDALL	$v.addAll(c)$	$\{\phi\}[v.elem, c.elem, (ap_i.elem, q)][v.elem, c.elem, (ap_j.elem, r)] \ldots$ $[v.elem, c.elem, (ap_m.elem, u)]$, where $\{(ap_i.elem, q), \ldots (ap_m.elem, u)\} = SubAPs^*(\phi)$, and $\phi[v.elem, c.elem, (ap_i.elem, j)]$ is as defined above in COLLECTIONADD.
INIT	$v.\langle init \rangle()$	*Same transfer function as* $v.clear()$.
PARAMETERISEDINIT	$v.\langle init \rangle(c)$	$\phi[c.elem/v.elem]$,
REMOVE	$v.remove(o)$	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 that that contain $v.elem$ as a sub access path.
LISTGET	$v = c.get(i)$	$\phi[c.elem/v]$

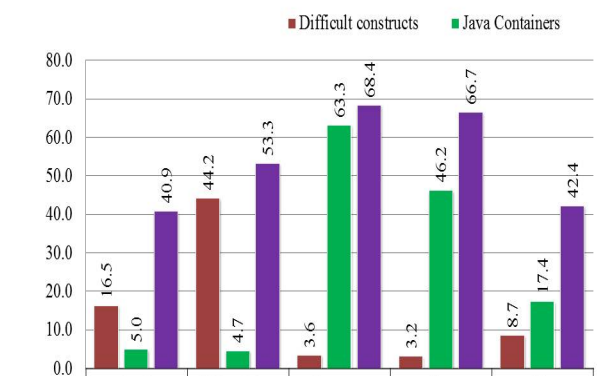
Figure 4.3: Transfer Functions for Java collections API methods

Chapter 5

Experimental Results

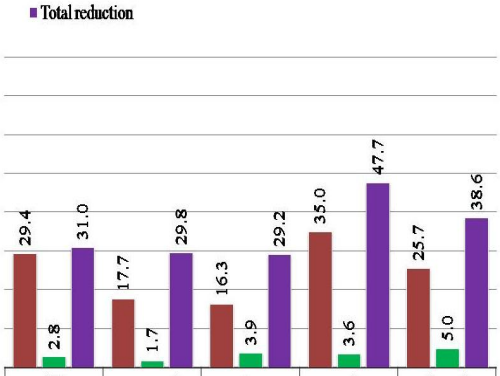
We use 10 real-world Java programs for evaluating our approach; these are the same programs used by Madhavan et al. [15] to evaluate the base analysis. We also additionally use the variant of the *Jolden* benchmarks created by Marron et al. [16]. The original *Jolden* benchmarks are based on the *Olden* C-language benchmarks, and are pointer-intensive programs. Marron et al. have 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.

In Wala [27] (which is the program analysis framework we use), to construct a call-graph for a program, one or more *entry methods* for the program need to be specified. We have specified all the *main* methods in each benchmark as the entry methods, except the main methods in test-suites. 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. 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 call graph, *may points-to*, *mod-ref*, and immediate-producers information that is pre-computed and provided by Wala. 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.



	bcel	Jbidwatcher	javacup	Jlex	sablecc
Total Deref	10143	9643	2851	2510	14017
Base Unsafe	1221	1652	608	93	2139
Ext Unsafe	722	771	192	31	1233

(a) Smaller benchmarks



	l2j	progaurd	ourtunes	antlr	freecol
	36899	18275	16449	17409	24077
	6015	2781	1532	4042	6994
	4149	1953	1085	2113	4293

(b) Larger benchmarks

Figure 5.1: Reduction in unsafe dereferences (%-age) when our analysis is run in different modes

5.1 Overall results

Figure 5.1 gives an overview of our results on the 10 real-world Java programs. The results for the five smaller benchmarks are shown in part (a), while part (b) is about the five larger benchmarks. For each benchmark we show three bars, each representing the percentage reduction in the number of *unsafe* dereferences reported by our analysis vis-a-vis the base analysis, when our analysis is run in three different modes. The first two modes correspond to turning on our two optimizations *individually*; i.e., the first mode corresponds to our *difficult-construct* optimization (Chapter 3) alone turned on, while the second is with our Java collections transfer functions (Chapter 4) turned on. The final bar shows the result with both our optimizations turned on; i.e., it corresponds to our full extended analysis. The table below the bars indicates certain absolute numbers for each benchmark: the total number of dereferences analyzed, the number of dereferences reported unsafe by the base analysis, and finally, the number of

dereferences reported unsafe by the extended analysis. We analyze every dereference in every non-library method in each benchmark, except dereferences to the variable *this* (which are always guaranteed to be safe).

The percentage of dereferences reported as unsafe by the extended analysis ranges from 1.2% in Jlex to 17.8% in freecol. On average, the extended analysis classifies about 9% of the dereferences in each benchmark as unsafe, vis-a-vis 16.3% for the base analysis. That is, on average per benchmark, the number of unsafe dereferences reported has come down by about 45%. Our results are even better on the *jolden* benchmarks; on average the number of unsafe dereferences reported per benchmark is down by 63% when compared to the base analysis (we provide more details about these results later).

We are not able to claim theoretically that the extended analysis analysis will report as safe *every* dereference that the base analysis analysis reports as safe. However, we found empirically that this desirable property actually holds on *every* single dereference in the ten real benchmarks that we have studied.

A metric that's related to precision is *average propagation count*. For any benchmark program, this metric is an average across all dereferences of the longest path of instructions through which propagation happens during the analysis of a dereference. We find that (a) In seven out of ten real 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 analysis continues propagating are frequent. (b) Across all ten benchmarks there is a general correlation between percentage change in propagation count and percentage increase in precision. That is, propagating along longer paths results in higher precision. We present the details of this study in Appendix E.1.

The increase in propagation counts has caused a corresponding increase in running time of the analysis. The extended analysis takes 427 millisecs on average to verify a dereference (over all ten real benchmarks), compared with 288 millisecs for the base analysis. The running time is still very reasonable, even in a demand-driven setting where quick response time is expected. Furthermore, we have empirical evidence that the extended analysis analysis spends extra time

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	1121	344	201	191	61
jbidwatcher	9643	1652	914	731	490	78
l2j	36899	6015	2367	1771	235	166
proguard	18275	2781	1243	493	113	46
ourtunes	16449	1532	253	250	97	59
javacup	2851	608	24	22	410	385
sablecc	14017	2139	265	186	827	373
jlex	2510	93	10	3	55	43
freecol	24077	6994	2167	1796	1302	353
antlr	17409	4042	2495	1496	456	146

Figure 5.2: Improvement due to optimizations to handle difficult constructs and Java collections

only on the dereferences that the base analysis analysis calls unsafe; the average analysis time per dereference has increased by only 1% for the dereferences that the base analysis reported as safe.

We now discuss each of our two optimizations *individually*, in order to bring out the potential applicability of each optimization, and also to bring out the extent to which the optimization meets its potential.

5.2 Using immediate producers to skip difficult constructs

Columns 1-5 in Figure 5.2 give an overview of the improvement in the precision of the extended analysis over the base analysis when we turn on *only* our optimization of using immediate producers to skip difficult constructs. Columns 2 and 3 of this figure are basically the same as the Rows 1 (Total Derefs) and 2 (Base unsafe) of the table shown below the bar graphs in Figure 5.1. In Column 4, 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 Chapter 3).

These are the dereferences that can potentially benefit from our optimization. In Column 5, we report the dereferences among the ones included in Column 4 that are proved safe by our optimization. This ratio of Column 5 over Column 3 is shown as a percentage by the first bar for every benchmark in Figure 5.1.

As evidenced by Columns 4 and 5 in Figure 5.2, on average across all benchmarks, we are able to prove 68.63% of the dereferences reported in Column 4 as safe. This ratio, in general, cannot be a 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 difficult-construct optimization does not address, and which cause a disjunct to get validated, hence resulting in the dereference being called unsafe. In general, this is also the reason why our two optimizations work synergistically, and give maximum benefit when turned on *together*. Note that in Figure 5.1 the third bar in each benchmark, except *l2j*, is taller (sometimes significantly so) than the sum of the heights of the first two bars (the second bar indicates gain in precision when only our collections optimization is turned on).

In spite of the empirical results suggesting that the idea of using immediate producers at difficult constructs is much more precise than the base analysis with its limits, we cannot claim that this will necessarily be true in all cases. We discuss this further in Appendix E.2.

We also evaluated the possibility of simply removing the limits in the base analysis regarding difficult constructs, and letting the analysis enter and analyze difficult constructs, rather than employ our optimization. For this specific study we excluded the benchmarks *antlr* and *freecol*, because the base analysis does not scale to them without the limits. The base analysis without the limit for virtual calls (see Chapter 3) needs *37 times* more time on average to verify a dereference per benchmark than when the limit is enforced. Yet, in this scenario, the precision gain is very low – reducing unsafe dereferences reported by only 3.4%. Similarly, removing the limits of analyzing library call-backs increases the average time to verify a dereference per benchmark by 3 times, while decreasing the unsafe dereferences reported by only 0.27%. Our difficult-construct optimization, on the other hand, reduces unsafe dereferences by 20.24%, while increasing average time to verify a dereference per benchmark by

only 2.5 times (considering all ten real benchmarks). These results clearly show the value of our optimization.

5.3 Handling Java Containers

In this section, we will discuss the effect of turning on *only* our optimization to handle Java collections methods. Column 6 in Figure 5.2 shows the number of unsafe dereferences reported by the base analysis during the analysis of which a root predicate is reduced to $true$ when the analysis is 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. Column 7 shows the number of dereferences in Column 3 (dereferences reported unsafe by base analysis) that were proved safe due to our technique to handle Java collection methods. The ratio of Column 7 over Column 3 is shown as a percentage in the second bar for every benchmark in Figure 5.1. Note that imprecision in the analysis of collections methods is the major cause of imprecision (among the two causes that we focus on) in three programs – *javacup*, *jlex*, and *sablecc*.

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. In the benchmarks *sablecc*, *progaurd*, *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 numbers in Column 6, respectively. In the *jbidwatcher*, we do not observe much reduction in the dereferences being reported unsafe earlier. We have already discussed in Section 5.2, why *all* the dereferences in a category of imprecision in general cannot be proven safe just by turning on the optimization for that category.

We also ran our analysis on the variant of the Jolden micro benchmarks that we introduced earlier. We only used 6 of the 9 benchmarks, as the others did not have any usage of collections. We used these micro benchmarks to check the effectiveness of our collections-technique in the absence of complexities that larger object-oriented programs generally exhibit. Figure 5.3 shows the results in each of the benchmarks. Columns 2, 3, 6, and 7 have the same meanings

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 5.3: Our results on Jolden benchmarks

as the corresponding columns in Figure 5.2. Note that the numbers in Column 7 were obtained with only our optimization for Java collections turned on. Note that on benchmarks *bh*, *mst* and *em3d*, our optimization 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 optimizations on. In the six micro benchmarks 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 optimizations are more effective on the micro benchmarks compared to the real world larger benchmarks.

Chapter 6

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 the optimizations that we propose to address, namely, using immediate producers to handle difficult constructs, and using manual summaries to handle Java collections methods, respectively.

6.1 Null dereference analysis

Xylem [17] and *Snugglebug* [2] are two previous approaches that are closely related to the base analysis, in the sense that they compute weakest pre-conditions 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 fixpoints. 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], which appeared in the literature after the publication of the base

analysis, 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 is discussed in Appendix A.

Salsa [14], and approach of Spoto et al. [24] 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 [15].

Shape analysis [20] 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 [13, 19, 7] for performing shape analysis by backward analysis. Of these, only the paper by Gulavani et al. [7] 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. [28] has been shown to scale to real programs (device drivers) of size approximately 10 kLOC, taking on average 430 seconds to analyze each benchmark.

6.2 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. [26] propose a technique to perform static *taint analysis* on Java programs to detect security vulnerabilities in web applications. Geay et al. [5] 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. [8] use a thin slice to obtain static path conditions for paths in Java programs. 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 3.4, 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; the basic idea would be to check if any statement that explicitly copies $null$ into a memory location is a source statement for the dereference. We also discussed why this technique would result in very low precision in our setting. 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).

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. [29], 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.

6.3 Java collections

There exist earlier approaches [16, 11, 4, 1, 18] 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 approach for heap analysis of Java programs by Marron et al. [16] 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; this is similar to our technique of

using the special field $elem$. 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 technique proposed by Gregor et al. [6] 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. [1] also propose a technique to verify if client code uses STL correctly. They use predicate-abstraction based model checkers for their analysis. However, none of the above techniques focuses on reasoning about the *contents* of containers, which is our focus.

The approach of Dillig et al. [4] 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 [9], 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.

The *Hob* verification framework [12, 11] can be used to verify whether client code that uses data structures satisfies the pre-conditions of these data structures, as specified by the data-structure implementations. Their approach addresses not just standard (library-provided) data structures, but also user-provided data structures. They do have a major limitation, namely,

that programs (both clients as well as the data structures) be implemented in their own programming language. Also, this language does not support difficult to analyze features such as inheritance, dynamic dispatch and object-based encapsulation, which are prevalent in Java programs. Furthermore, they have evaluated their approach only on programs measuring up to about 2000 LOC. However, the kind of properties they address are more deep that just null-ness properties.

Recently, Parízek et al. [18] 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, translate the program into an abstract boolean program, and then model-check this boolean program. They specify weakest pre-conditions (WP) rules for the Collections API methods, and interpret these rules using an SMT solver in order to create the abstract boolean program. 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.

It is noteworthy that the approaches mentioned above are all based on forward analysis, and hence, are not suitable in a demand-driven setting. Our approach as well transfer functions are quite different from the ones employed in these approaches. We refer the reader to our discussion in Section 4.4 about this aspect.

Appendix A

Details of the inter-procedural aspect of the base analysis

The inter-procedural aspect of the base analysis is modeled on that of Xylem [17], which itself is based on Sharir and Pneuli's tabulation based approach [21]. When a callsite to a method m is encountered in a caller, the analysis propagates the disjunct ϕ at the point after the callsite to the end of m's body. The propagated disjunct is generated by replacing all the actual arguments in ϕ by corresponding formal parameters of m. This disjunct is propagated up through m (and it's transitive callees). The set of disjuncts that results at the entry of m is then propagated back to the point before the callsite (with appropriate renaming). When a virtual callsite is encountered, pre-computed may-points-to information is used to resolve the potential targets of the call.

When the root dereference is in a method m, or in a callee of m, and the propagated disjuncts reach 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; we call the methods that contain these call-sites the *predecessors* of m.

The analysis follows a *depth-first* propagation, rather than a chaotic strategy. That is, when a disjunct reaches a call-site in a caller, the analysis of the caller is suspended until analysis of the callee and all its transitive callees is complete. This improves the space-efficiency of the analysis, because CFGs and intermediate analysis results are kept in memory only for the

stack of method calls currently being analyzed. The analysis uses a novel fix-point strategy for analysis of recursive and mutually recursive methods in a sound manner as part of this depth-first traversal (the details of which we omit, for the sake of brevity).

For the sake of efficiency, as well as to ensure termination, the analysis maintains a summary table $\Sigma[[m]] : Disjunct \rightarrow 2^{Disjunct}$, 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 (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 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.

Appendix B

Some details about our handling of virtual calls and library calls

The base analysis employs several heuristics to limit the analysis of library methods, whether they are invoked directly or via virtual calls. This is because analysis of library methods is typically expensive, and lacking in precision. These limits are employed even when the library method is invoked via a virtual call and the call has fewer targets than the user-specified threshold (see Section 3). Firstly, the base analysis uses a manually provided *skip list*, which contains library methods that are known not to have side-effects. Methods in this list are entered and analyzed only if the return value from the call is used in the post-condition. The base analysis also uses other heuristics as filters, which cause certain target library methods (with side effects) to be never analyzed; rather, these targets are handled conservatively, by reducing to $true$ (i.e., dropping) all predicates in the post-condition that include access-paths that may be modified by the target (as per a pre-computed mod-ref analysis). In the extended analysis, whenever *any* target of a virtual call that is not in the skip list is filtered out by these heuristics we elect to not enter *any* of the targets of the call, and also, to not drop any affected predicates from the post-condition; instead, we apply procedure *IsInvalidUsingTS* on the post-condition after the call, and reduce the post-condition to $true$ or $false$ based on the return value of *IsInvalidUsingTS*, as described in Section 3.3.

Note that there are two distinct scenarios in which our analysis encounters library methods:

(1) The root dereference is in a call-back method, and disjuncts need to be propagated from the entry of the call-back method to call-sites to the call-back method inside library methods, and (2) we have a post-condition at the point after a library call in application code. The heuristics in the base analysis that we mentioned above are applicable only in Scenario (2), because they are based simply on sending a pruned version of the post-condition to the point that precedes the library call. These heuristics are not applicable in Scenario (1), wherein we need to propagate a disjunct from the entry of a call-back method m to points in the library methods L where m is called, and then eventually to points in the client code where library methods in L are called. We target this issue in the extended analysis using the technique mentioned in Section 3.

Appendix C

Details about certain transfer functions in Figure 4.3

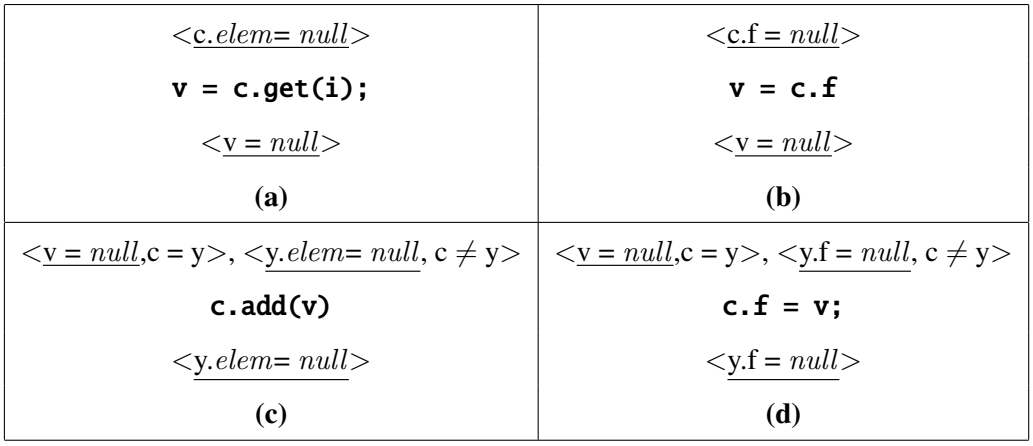


Figure C.1: Illustration of the similarities between the transfer functions of (a) LISTGET and (b) GETFIELD, and (c) COLLECTIONADD and (d) PUTFIELD.

The LISTGET rule resembles the GETFIELD rule in Figure 2.2. This similarity is illustrated in Figure C.1, parts (a) and (b). Similarly, the COLLECTIONADD rule bears some similarity with the PUTFIELD rule in Figure 2.2. This similarity is illustrated in Figure C.1, parts (c) and (d). This figure also illustrates how Rule COLLECTIONADD embeds *aliasing* predicates in the disjuncts that it produces in the pre-condition, just like the PUTFIELD rule. These aliasing predicates can get invalidated later, and can hence increase precision.

The COLLECTIONADDALL rule is for the method-call $v.addAll(c)$, which adds all the elements of the collection c to collection v. This is very similar to the COLLECTIONADD transfer function.

The COLLECTIONCLEAR 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 predicates. 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.

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 D

Implementation Details

D.1 Analysis Framework

We have implemented our approach using the Wala [27] program analysis framework. Wala provides us control-flow graphs of methods, as well as pre-computed may-points-to information. We use a flow-insensitive and receiver-type context sensitive mode for the points-to analysis (namely, `ReceiverTypeContextSelector`), where the context is based on the concrete type of the receiver object. We also use the inexpensive context-insensitive thin slicer provided by Wala to determine the immediate producers of statements. 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 2.2, to reduce the number of aliasing combinations that need to be considered, and (d) in the COLLECTIONADD and COLLECTIONADDALL rules (see Figure 4.3), for a similar purpose as in the PUTFIELD rule. Imprecision in Wala's points-to analysis affects the precision of our approach due to points (a) and (b) above, and affects the scalability of our approach due to all four points above. Note that points (c) and (d) do not directly affect our precision, because we anyway model aliasing relationships explicitly. 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

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 ϕ' with *(ap,*s_i*)*

Step 4: **END FOR**

Step 5: **RETURN** ϕ'

Figure D.1: Pseudocode to update originating statement of an access path

but highly scalable. In the rest of this chapter we describe a few interesting details about our implementation.

D.2 Immediate producers

In Section 3.2 we used a routine *ImmProds(ap,p)* (see Figure 3.3), which is supposed to return the immediate producers of an access path *ap* at a program point *p*. We find these immediate producers using the thin slicer implementation provided by Wala, as was pointed out earlier. This implementation, however, can only return the immediate producers of a statement, and not those of an access path at a program point. 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` at the point preceding the statement (but not the immediate producers of `v`). Our approach to implementing *ImmProds* is to 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. We track the originating statement because of the property that the immediate producers of an access path in a formula at any program point are *same* as the immediate producers of the originating-statement associated with that access path.

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

Figure D.2: Modified transfer functions for tracking *originating* statement

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 2.2) to incorporate originating-statement information into access paths. Figure D.2 shows these modified transfer functions, while Figure D.1 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, $APs((v.f, s_n) = null)$ 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 $\langle (v, s_j) = null, (v.f, s_k) = null \rangle$, the precondition will be $\langle (w, s_i) = null, (w.f, s_k) = null \rangle$. 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 optimization to skip the difficult construct.

D.3 Identifying collection method calls safely

Consider a call `c`.*foo*(), where the declared type of `c` is one of the standard Java collections interfaces. It would be safe to use the transfer function for *foo* in Figure 4.3 only if the analysis is certain that `c` does *not* point to any object of a user-defined type that implements a standard collections interface. That is, `c` should potentially point only to objects of the standard collections classes. Otherwise, the call would need to be handled as a regular call (i.e., by entering and analyzing it). We check this using our utility predicate $IsCollectionCall(c.foo(\ldots))$, which we define in Figure D.3. The function $PointsTo(v)$ returns the set of symbolic objects that may be referred to by the access path v. The function $GetConcreteType(SymObj)$ returns the concrete type of the concrete objects represented by the symbolic object $SymObj$.

$LibrarySubClasses(ifc)$ returns all the library classes implementing the interface ifc.

$IsCollectionCall(c.foo(\ldots))$ = $\forall SymObj \in PointsTo(c).(GetConcreteType(SymObj)$
$\in (LibrarySubClasses(java.util.Set) \cup LibrarySubClasses(java.util.List) \cup$
$LibrarySubClasses(java.util.Iterator)))$

Figure D.3: Identifying calls to standard collections methods

D.4 New simplification rules to invalidate infeasible disjuncts

We use several additional simplification rules in our extended analysis over the base rules shown in Figure 2.3, 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 $this.f = null$. 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 *this.f* is infeasible.

- If an access path $null.f_1.f_2.f_3 \ldots f_n$ 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 2.2), 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 $ap.elem$ 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.

D.5 Missing call targets

It is possible that our analysis encounters virtual method calls that do not have any targets. This may happen if the target of a method call is defined in a class that is not available to the analysis. In most cases, these method calls are calls to the GUI and JDBC libraries, which we do not link, as this was observed to affect adversely the efficiency of Wala's points-to analysis. We treat such method calls conservatively. Predicates that use the return value of method calls that do not have a target are dropped. We also over- approximate the side-effects of such a method call by assuming that it may modify every object reachable from the arguments passed to it. We use this over-approximate side effects information to drop predicates that may be modified by the method call.

Appendix E

Additional experimental results

E.1 Correlation between propagation count and precision

The *propagation count* of a dereference refers to the longest path of instructions through which propagation happens 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 disjuncts from a given input disjunct, the generated disjuncts get a count which is one more that of the input disjunct (a summary-hit is treated as a propagation through a single instruction). 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 propagation count of a benchmark is the average propagation count of all its dereferences. For each of our ten benchmarks, the first bar in Figure E.1 shows the factor of change in the propagation count for the extended analysis over the base analysis. The second bar shows the percentage reduction in unsafe dereferences for the extended analysis over the base analysis; here, a taller bar indicates more gain in precision (this bar is a reproduction of the third bar in Figure 5.1 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 analysis continues the analysis are frequent. (b) Across

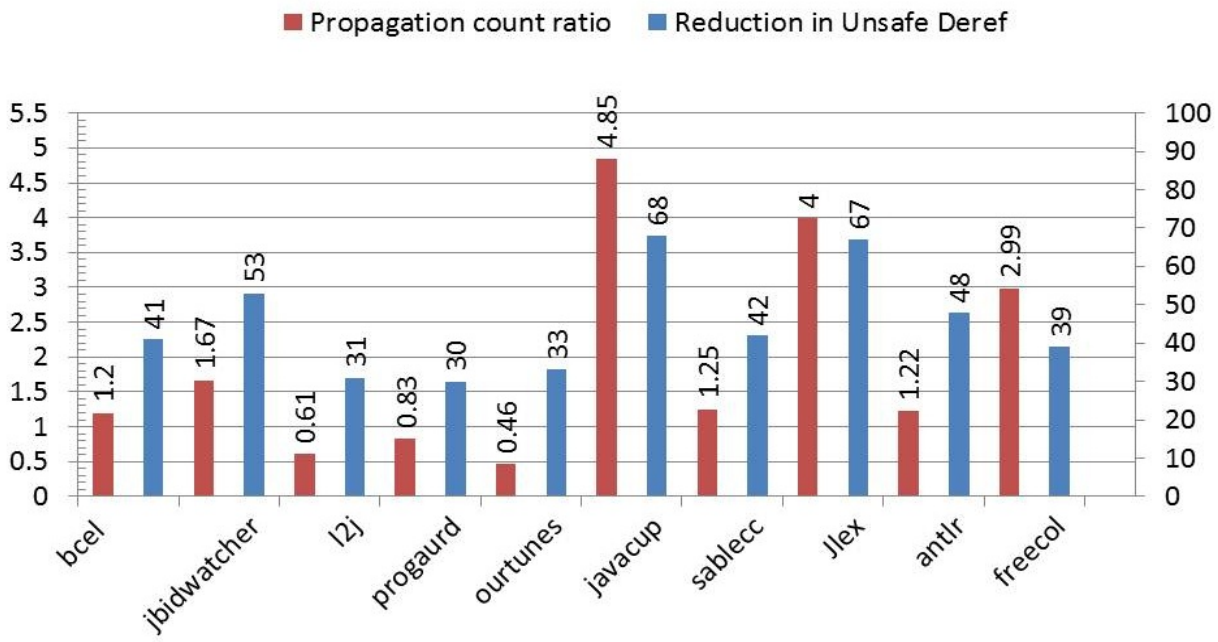


Figure E.1: Increase in precision and in propagation count of the extended analysis. The left y-axis indicates the ratio of average propagation count of the extended analysis over the base analysis, while the right y-axis indicates the corresponding reduction in percentage of unsafe dereferences.

Code	Base	Extended
1: r.f = null;	$false$	$true$
2: v = new A();	$false$	
3: z.init();	$\langle \texttt{v} = null \rangle$	
4: if(v == null)	$\langle \texttt{z.f} = null, \texttt{v} = null \rangle$	$\langle \texttt{z.f} = null, \texttt{v} = null \rangle$
5: print **z.f.g**;	$\langle \texttt{z.f} = null \rangle$	$\langle \texttt{z.f} = null \rangle$

Figure E.2: Illustration to demonstrate imprecision in our idea of using immediate producers.

all ten benchmarks there is a general correlation between change in propagation count and increase in precision. This confirms our expectation that if we reduce the situations where we give up early we get higher precision. (c) In the benchmarks *l2j, proguard* and *ourtunes* the extended 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 optimizations actually reduce the propagation count (because, e.g., we do not enter and analyze a library call or targets of a virtual call).

E.2 Precision and efficiency due to our immediate-producers optimization

In spite of the empirical results suggesting that idea of using immediate producers at difficult constructs is much more precise than the base analysis with its limits, we cannot claim that this is always necessarily the case. We demonstrate this using the example in Figure E.2. Say the call-site to the method `init()` in Statement 3 resolves to too many methods; also, say that due to imprecision in Wala's construction of the immediate producer relation Statement 1 is indicated as an immediate producer of the access path `z.f` at Statement 5. The base analysis and the extended analysis begin at the root dereference (of `z.f`) at Statement 5, and generate the same disjunct at the point above Statement 4 as shown to the right of this statement. Now, rather than analyze the call in Statement 3 (which has too many targets) the base analysis treats the root dereference "$\texttt{z.f} = null$" as an *affected predicate*, and drops it (i.e., reduces it to $true$).

However, the other predicate in the disjunct, i.e., "$\mathtt{v} = null$" gets invalidated at Statement 2, resulting in the root dereference being called safe. On the other hand, at Statement 3, the extended analysis drops the non-root predicate $\mathtt{v} = null$, and carries the root predicate to its (infeasible) immediate producer Statement 1; this causes it to declare the root dereference unsafe.

Similarly, in spite of the clear empirical evidence regarding the efficiency of our extended analysis vis-a-vis the base analysis with no limits, we cannot claim that this will necessarily hold in all cases. Due to the lack of path sensitivity in our approach of fetching the immediate producers of an access path, our analysis may imprecisely consider a statement to be one of the immediate producers, and then do analysis from it. The same statement may never be analyzed by the base analysis due to its limited-path sensitivity. Under such scenarios the base analysis, even without its limits, can be more efficient than our extended analysis.

Appendix F

Proof of Soundness of Transfer functions

In this subsection we will consider a few representative abstract transfer functions from Figure 4.3 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 2.1, and then extended in Figure 4.1.

F.1 Concrete Lattice

$\eta \in C = 2^{State}$	: 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
$Addr$ = $Loc \times (Offset \mid \epsilon)$	: 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 F.1: Concrete Semantics

We first introduce in Figure F.1 the concrete lattice that we use to specify the concrete

semantics of the collections APIs. It is inspired by the *standard* memory model described by Sotin et al. [23]. 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, while an 'ϵ' is used while referring to a scalar cell. The $o \in Offset$ can either be a declared program field ($FieldName$) (used to refer to a field of a non-collection object), or a value belonging to the natural numbers $\mathbb{N}$ (used to refer to an element of a collection object). All collection objects are modeled in a uniform way in our concrete domain. A collection object has natural numbers as offsets that refer to its elements. The offset 0 refers to the first element of 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 that refers to 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.

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 $v.f_1.f_2...f_n$ is retrieved by the following recursively defined macro:

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

A concrete state (e, s) is said to satisfy a formula ϕ, written as $(e, s) \vdash \phi$, if the formula 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 $s(e(v), \epsilon)$,
2. else if x is of the form $v.elem$, substitute x with $s(s(e(v), \epsilon), o)$, for any natural number o,

3. else if x is of the form $v.collection$, substitute $v.collection$ 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 $GetContents(e, s, v, f_1, f_2, \ldots, f_n)$.

The concretization function γ: $Formula \rightarrow 2^{State}$ is defined as:
$\gamma = \lambda\phi \in Formula. \{(e, s) \mid (e, s) \in State \wedge (e, s) \vdash \phi\}$.

F.2 Proofs

We will now prove the correctness of certain representative abstract backwards transfer functions from Figure 4.3. For each abstract backwards transfer function f_d that we consider in this section we show (a) the transfer function itself, from Figure 4.3, (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. 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.next()*

$f_d = \lambda\phi.\, \phi'$, where $\phi' = \phi[i.collection.elem/v]$.

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) = $(GetContents(e, s, i, c), GetContents(e, s, i, next))$

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) = $s[(s(e(i), \epsilon), next) \mapsto (GetContents(e, s, i, next) + 1)]$

Forward concrete function:

$f_c{}^f$ = $\lambda\eta.\ \{(e, s_2) \mid (e, s) \in \eta \wedge$

$hasNext(e, s, i) \wedge s_1 = s[(e(v), \epsilon) \mapsto s(nextAddr(e, s, i)] \wedge s_2 = incNext(e, s_1, i)\}$

Backward concrete function:

f_c^b = $\lambda\eta.\ \{(e, s) \mid (e, s') \in \eta \wedge hasNext(e, s, i) \wedge s_1 = s[(e(v), \epsilon) \mapsto s(nextAddr(e, s, i)]$

$\wedge\ s' = incNext(e, s_1, i)\}$

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

$f_c^b(\gamma(\phi))$ = $f_c^b(\{(e, s) \mid (e, s) \in State \wedge (e, s) \vdash \phi\})$

= $\{(e, s) \mid hasNext(e, s, i) \wedge (e, s') \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto s(nextAddr(e, s, i))]$

$\wedge\ s'$ = $incNext(e, s_1, i)\ \}$

$\subseteq \{(e, s) \mid (e, s') \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto s(nextAddr(e, s, i))] \wedge s'$ = $incNext(e, s_1, i)\}$

$\subseteq \{(e, s) \mid \exists j \in \mathbb{N}.\ ((e, s') \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto GetContents(e, s, i, c, j)]$

$\wedge\ s'$ = $incNext(e, s_1, i))\}$

Now, since ϕ does not refer to the *next* fields of iterators, for any stores s' and s_1 such that $s' = incNext(e, s_1, i)$, and for any formula ϕ, $(e, s') \vdash \phi$ iff $(e, s_1) \vdash \phi$.

Therefore, we get

$f_c^b(\gamma(\phi)) \subseteq \{(e, s) \mid \exists j \in \mathbb{N}.\ ((e, s_1) \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto GetContents(e, s, i, c, j)]\}$

Now, from substitution Rules 2 and 3 mentioned in Appendix F.1, it follows that $\exists j\ \in$

$\mathbb{N}.\ ((e, s_1) \vdash \phi \wedge s_1 = s[(e(v), \epsilon) \mapsto GetContents(e, s, i, c, j)])$ iff $(e, s) \vdash \phi[i.collection.elem/v]$.

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) $v.add(w)$

f_d = $\lambda\phi.\ \phi'$, where ϕ' = $\phi[v.elem, w, \langle ap_1.elem, 1\rangle] \ldots$
$[v.elem, w, \langle ap_2.elem, 1\rangle] \ldots$
$[v.elem, w, \langle ap_n.elem, 1\rangle] \ldots [v.elem, w, \langle ap_n.elem, m\rangle]$,
where $\{(ap_1.elem, 1), \ldots (ap_n.elem, m)\} = SubAPs^*(\phi)$, and $\phi[v.elem, k, \langle ap_i.elem, j\rangle]$ is as defined in rule COLLECTIONADD in Figure 4.3.

v 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 v refers to a set.

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

$CanbeAdded(e, s, set, elem)$ checks if we are allowed to add the element referred to by variable $elem$ to the collection referred to by variable set in state (e, s). We omit the details of this function, because it depends on the exact kind of set object that set refers to (e.g., normal set, multi set, etc.)

$added(e, s, s', v, w)$ checks if (e, s') is the state obtained by adding the object pointed by w to some offset in the collection pointed to by v in the state (e, s). It is formally defined as

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

Forward concrete function:
$f_c{}^f$ = $\lambda\eta.\ \{(e, s') \mid (e, s) \in \eta \wedge ((CanbeAdded(e, s, v, w) \wedge added(e, s, s', v, w)) \vee (!CanbeAdded(e, s, v, w)$
$\wedge\ s' = s))\}$

The concrete transfer function ensures that the element to be added gets added at the lowest "available" offset in the collection; the details of this have been omitted above for brevity.

Backward concrete function:

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

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

$f_c^b(\gamma(\phi)) = f_c^b(\{(e,s) \mid (e,s) \in State \wedge (e,s) \vdash \phi\})$

$= \{(e,s) \mid ((((e,s') \vdash \phi) \wedge CanbeAdded(e,s,v,w) \wedge added(e,s,s',v,w)) \vee (!CanbeAdded(e,s,v,w) \wedge ((e,s) \vdash \phi)))\}$

$\subseteq \{(e,s) \mid ((((e,s') \vdash \phi) \wedge added(e,s,s',v,w)) \vee ((e,s) \vdash \phi))\}$, by removing some conjuncted conditions. ***– (i)***

We define a utility function $Transform$, which takes a parameters e,s,ϕ,v,w, such that v refers to a collection in (e,s), w refers to any object, and $(e,s) \vdash \phi$. This function returns a new formula ϕ^t, by following the steps:

(a) Find a *maximal* subset η of $SubAPs^*(\phi)$ such that (a) each access path ap_i in η is such that ap_i ends with a '$.elem$' field, ap_i and v refer to the same (collection) object in (e,s), and one of the elements of this collection object is the object pointed to by w in (e,s), and (b) if we replace each access path in ϕ that's in η by w the resulting formula ϕ^t is satisfied by (e,s).

(b) Return ϕ^t.

Lemma 1: Say (e,s) and (e,s') are two states such that (a) they are identical except that the collection objects referred to by a variable v in (e,s) and in (e,s'), respectively, differ in their contents, and (b) these two collection objects differ at a single offset i, and at this offset the collection in (e,s') contains the same element as the one that w points to in (e,s'). Then, for any formula ϕ, the following result holds:

$((e,s') \vdash \phi) \Rightarrow ((e,s') \vdash Transform(e,s',\phi,v,w)) \Rightarrow ((e,s) \vdash Transform(e,s',\phi,v,w))$.

Proof:

From the definition of the function $Transform$ it is easy to see that $((e, s') \vdash \phi) \Rightarrow ((e, s') \vdash Transform(e, s', \phi, v, w))$. To prove the second implication in the lemma, we assume that $(e, s') \vdash Transform(e, s', \phi, v, w)$, and argue that it is hence not possible that $(e, s) \nvdash Transform(e, s', \phi, v, w)$. Since s and s' differ only in the content stored at offset i of the collection pointed to by v, only predicates that involve an access path $ap_i.elem$, where ap_i aliases with v, can potentially be not satisfied by (e, s). Consider such a predicate. Since it is satisfied by (e, s'), and (e, s) and (e, s') differ only at offset i in the collection pointed to by v, it must be that this predicate is satisfied by (e, s') using the element at offset i. However, (a) this element is pointed to by w in (e, s), and (b) by the definition of $Transform$, this predicate must have been rewritten to involve w (rather than $ap_i.elem$) in the formula $Transform(e, s', \phi, v, w)$. Therefore, this (rewritten) predicate must be satisfied by (e, s). Therefore, it is not possible for $(e, s) \nvdash Transform(e, s', \phi, v, w)$. Hence proved.

Now, applying Lemma 1, we can rewrite (i) as

$f_c^b(\gamma(\phi)) \subseteq \{(e, s) \mid (((\underline{((e, s) \vdash Transform(e, s', \phi, v, w))} \wedge added(e, s, s', v, w)) \vee ((e, s) \vdash \phi))\}$, where the changed portion is underlined.

Given that $\phi' = f_d(\phi))$, by the definition of $Transform(e, s', \phi, v, w)$ it is easy to see that $Transform(e, s', \phi, v, w) \Rightarrow \phi'$. Therefore, we get

$f_c^b(\gamma(\phi)) \subseteq \{(e, s) \mid (((e, s) \vdash \phi' \wedge added(e, s, s', v, w)) \vee ((e, s) \vdash \phi))\}$.

Now, we drop the conjunct $added(e, s, s', v, w))$. Also, since $\phi \Rightarrow \phi'$, we replace ϕ in the second disjunct in the formula above with ϕ'. Upon simplification we get

$f_c^b(\gamma(\phi)) \subseteq \{(e, s) \mid ((e, s) \vdash \phi')\ \}$.

Therefore, by definition of $\gamma(f_d(\phi))$, we get

$f_c^b(\gamma(\phi)) \subseteq \gamma(f_d(\phi))$.

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 ϕ' = $\phi[v/i.collection]$

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 F.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's $true$ 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).

The concrete transfer function above ensures that all elements in collections in the post-states (e,s) occupy contiguous offsets starting from zero; the details of this have been omitted above for brevity.

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) :

$(matchedEntry(e,s,s',v,i)))\}$.

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

$f_c^b(\gamma(\phi)) = f_c^b(\{(e,s) \mid (e,s) \in State \wedge (e,s) \vdash \phi\})$

$= \lambda\eta.\ \{(e,s') \mid (e,s) \vdash \phi \wedge IsCollection(v) \wedge (\forall i.\ ((equals(s'(s'(e(v),\epsilon),i), s'(e(w),\epsilon)))$

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

– *(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) $GetContents(e, s', v, i) = GetContents(e, s, v, i)$. 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.

Using Lemma 2, continuing from *(i)*, we get

$= \lambda\eta.\ \{(e, s') \mid \underline{(e, s') \vdash \phi} \wedge IsCollection(v) \wedge (\forall i.\ ((equals(s'(s'(e(v), \epsilon), i), s'(e(w), \epsilon)))$? (the address $(s'(e(v), \epsilon), i)$ is not mapped to any value in s) : $(matchedEntry(e, s, s', v, i)))\}$

(*the underlined part is the changed part*)

$\subseteq \lambda\eta.\ \{(e, s') \mid (e, s') \vdash \phi\}$ (*by eliminating some conjuncts*)

$\subseteq \gamma(f_d(\phi))$.

Bibliography

[1] N. Blanc, A. Groce, and 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, pages 521–524, New York, NY, USA, 2007. ACM.

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

[3] P. Cousot and 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, pages 238–252, New York, NY, USA, 1977. ACM.

[4] I. Dillig, T. Dillig, and 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, pages 187–200, New York, NY, USA, 2011. ACM.

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

[6] D. Gregor and S. Schupp. Stllint: lifting static checking from languages to libraries. *Software: Practice and Experience*, 36(3):225–254, 2006.

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

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

[9] L. Hendren. Scaling java points-to analysis using spark. In *Compiler Construction, 12th International Conference, volume 2622 of LNCS*, pages 153–169. Springer, 2003.

[10] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

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

[12] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying datastructureconsistency. In R. Bodik, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 137–138. Springer Berlin / Heidelberg, 2005.

[13] T. Lev-Ami, T. R. M. Sagiv, and S. Gulwani. Backward analysis for inferring quantified preconditions. In *Tel Aviv University Tech Report TR-2007-12-01*, 2007.

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

[15] R. Madhavan and R. Komondoor. Null dereference verification via over-approximated weakest pre-conditions analysis. In C. V. Lopes and K. Fisher, editors, *OOPSLA*, pages 1033–1052. ACM, 2011.

[16] M. Marron, D. Stefanovic, M. Hermenegildo, and 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, pages 31–36, New York, NY, USA, 2007. ACM.

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

[18] P. Parízek and O. Lhoták. Predicate abstraction of java programs with collections. In *OOPSLA '12: Proc. ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications*, pages 75–94, 2012.

[19] A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 314–327, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] M. Sagiv, T. Reps, and 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, pages 105–118, New York, NY, USA, 1999. ACM.

[21] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Application*. Prentice Hall Professional Technical Reference, 1981.

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

[23] P. Sotin, B. Jeannet, and X. Rival. Concrete memory models for shape analysis. *Electron. Notes Theor. Comput. Sci.*, 267:139–150, October 2010.

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

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

[26] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and 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, pages 87–97, New York, NY, USA, 2009. ACM.

[27] *T.J. Watson Libraries for Analysis (WALA), http://wala.sf.net*.

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

[29] W. Zhang and 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, pages 63–74, Washington, DC, USA, 2006. IEEE Computer Society.