Explaining Failures of Program Analyses *

Daniel von Dincklage
Department of Computer Science
University of Colorado
daniel.vondincklage@colorado.edu

Amer Diwan
Department of Computer Science
University of Colorado
diwan@cs.colorado.edu

Abstract

With programs getting larger and often more complex with each new release, programmers need all the help they can get in understanding and transforming programs. Fortunately, modern development environments, such as Eclipse, incorporate tools for understanding, navigating, and transforming programs. These tools typically use program analyses to extract relevant properties of programs.

These tools are often invaluable to developers; for example, many programmers use refactoring tools regularly. However, poor results by the underlying analyses can compromise a tool's usefulness. For example, a bug finding tool may produce too many false positives if the underlying analysis is overly conservative, and thus overwhelm the user with too many possible errors in the program. In such cases it would be invaluable for the tool to explain to the user *why* it believes that each bug exists. Armed with this knowledge, the user can decide which bugs are worth pursing and which are false positives.

The contributions of this paper are as follows: (i) We describe requirements on the structure of an analysis so that we can produce reasons when the analysis fails; the user of the analysis determines whether or not an analysis's results constitute failure. We also describe a simple language that enforces these requirements; (ii) We describe how to produce necessary and sufficient reasons for analysis failure; (iii) We evaluate our system with respect to a number of analyses and programs and find that most reasons are small (and thus usable) and that our system is fast enough for interactive use.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.3.2 [*Programming Languages*]: Language Classifications

General Terms Measurement, Reliability, Documentation

1. Introduction

Tools based on program analyses are useful for understanding and transforming programs. In the best case, these tools produce exactly the results that the programmer needs; e.g., the tool identifies the few lines of code that contain the source of a bug. In the worst case, these tools are ineffective; e.g., the tool tells the programmer that the bug could be anywhere in the program.

If program analyses produce undesirable results, the tool that depends on them may become ineffective. Prior work suggests two possible strategies for this. First, we can give up. Typically, compilers adopt this strategy: if they are unable to prove, using program analyses, that an optimization applies to a particular program location, they give up on the optimization for that location. Second, we can obtain guidance from the programmer; interactive optimization systems [6, 1] take this approach. The first strategy is acceptable especially for compiler optimizations (e.g., it would be cumbersome for a compiler to ask a user about hundreds of possible aliases). If the potential payoff is large enough, the second strategy is worthwhile; e.g., guidance from the user may eliminate a major bottleneck in the program. This paper focuses on the second strategy.

In order to obtain useful guidance from the programmer, the program analysis should be able to produce a *reason* every time it produces an *undesirable* result. The user of the analysis (e.g., interactive optimization system) provides a predicate that determines whether or not a given analysis result is desirable.[1] This reason should satisfy two requirements:

1. The reason should be *necessary*: the reason should only describe issues that contribute to undesirable results.
2. The reasons should be *sufficient*: the analysis should cease to produce undesirable results once we have addressed all the issues identified by the reason.

While some prior interactive optimization systems produce reasons for their failure, their reasons typically do not satisfy our requirements. Specifically, ParaScope [1] tells the user which dependences prevented it from parallelizing a loop. SUIF Explorer [6] goes further by also providing the programmer with program slices [13] of the array references involved in the dependences. While knowing those dependencies that inhibit the parallelization is useful to the user, the user does not know *why* the system believes that there is a dependence; thus dependencies alone are not *sufficient* reasons. Since slices may contain many statements that are irrelevant to the dependencies that inhibit parallelization, SUIF Explorer's slices are not *necessary* reasons. This paper describes and evaluates a novel approach for computing reasons that are necessary and sufficient.

The contributions of this paper are as follows:

- We describe the requirements an analysis must satisfy to enable us to produce reasons when the analysis produces undesirable

* This work is supported by NSF grant ST-CRTS 0540997. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

[1] In this paper we will use the phrase "analysis failed" synonymously with "analysis produced undesirable results".

results. We describe an *analysis language* that ensures that all analyses written in it satisfy these requirements. We show how to express data-flow analyses using our language and thus demonstrate that our requirements do not prevent us from at least expressing all data-flow analyses.

- We give two semantics for the analysis language. The *value* semantics defines how to evaluate analyses written in the language. The *reason* semantics defines how to compute reasons when the analyses produce undesirable results.
- We show that our approach produces necessary and sufficient reasons.
- We evaluate our prototype and demonstrate that it efficiently computes reasons for undesirable results. Moreover, we demonstrate that for the analyses that we tried, over 98% of all undesirable results have small reasons (3 or fewer terms). In other words, we do not overwhelm the programmer with complex reasons for each failure.

This paper is structured as follows. Section 3 describes what we mean by reasons. Section 4 describes our requirements for a language for writing analyses and also describes a language that satisfies our requirements. Sections 5 and 6 describe how we compute reasons for boolean and non-boolean analyses respectively. Section 7 discusses theoretical properties of our approach. Section 8 presents issues when using our system. Section 9 presents an example of using our system. Section 10 experimentally evaluates our system. Section 11 reviews related work and Section 12 concludes.

2. High-Level Example

Before we introduce the details of our approach, we illustrate the approach using an example.[2] In our example, a user asks an interactive optimizer to apply dead-store elimination to the store "a = x.f" in the code fragment below:

```
a = x.f;
...
q = a;
```

The optimizer uses the following analysis to determine whether it can apply the optimization to the store:

deadStore(store) := *hasNoUse*(store) ***or***
forall use in *usesOf*(store): *isDead*(use)

If this analysis returns "true" for a given store, the optimizer can safely eliminate the store. If it returns "false" the dead-store elimination is unsafe. Unfortunately, this analysis evaluates to "false" for our example; thus, the optimizer uses our approach to produce a reason for this failure which it communicates to the user.

To compute a reason, our approach records which properties contribute to the failure of the analysis. Each such property contributes to the overall reason for why the dead-store elimination is inapplicable.

Our analysis for dead-store elimination uses three properties: *hasNoUse*, *usesOf*, and *isDead*. The invocation of *hasNoUse(a = x.f)* evaluates to false and thus contributes to the analysis failure: If it had evaluated to true, the analysis would have succeeded. We therefore record "hasAUse(a = x.f)" as a failure reason. The invocation of *isDead(q = a)* fails since the assignment is not dead. We record "not-isDead(q = a)" as failure reason. Since the analysis combines these invocations with an "***or***", we construct the following reason:

OrReason(not-hasNoUse(a = x.f), not-isDead(q = a))

[2] For ease of exposition, this section uses simplified notation for reasons and analyses.

This reason explains to the user exactly why her requested optimization is inapplicable. The user can either address the reason (e.g., by asserting that g(a) is dead) to enable the optimization or decide that the optimization is indeed inapplicable.

3. What is a Reason

Before we can describe how we compute reasons for analysis failures, we need to precisely describe what we mean by reasons. A reason is a value of type $\mathcal{R}$. $\mathcal{R}$ contains two kinds of reasons: *root* reasons and *constructed* reasons.

A root reason says that the analysis failed because a property computed externally to our system is false.[3] We can add new root reasons to our system by providing it with new properties and the means to compute the properties.

Here are some root reasons we use in the examples in this paper (we introduce additional root reasons as convenient):

1. AnalysisFailed(*CannotModify*, s, v). The property *Modifies* with arguments s and v is false (i.e., statement s does not modify variable v).
2. AnalysisFailed(*canThrow*, s). The statement s may throw an exception.
3. *NoReason*. No reason needed since the analysis produced desirable results.

Constructed reasons combine reasons to indicate that the analysis failed due to a combination of reasons. We combine reasons using conjunction and disjunction, specifically, we use *AndReason(a,b) / OrReason(a,b)*, $a, b \in \mathcal{R}$, to combine reasons a and b.

If the programmer wants a failing analysis to succeed, she must resolve the reasons for the analysis's failure. A user can resolve a reason by: (i) for an AndReason, resolving both sub-reasons; (ii) for an OrReason, resolving either sub-reason; (iii) for a NoReason, doing nothing; and (iv) for a "AnalysisFailed(not-*property*,$\cdots$)" reason, taking appropriate steps to make *property*($\cdots$) hold.

As stated earlier, we require reasons to be *necessary* and *sufficient*. By necessary, we mean that resolving the reason guarantees that the analysis will be successful. By sufficient, we mean that if the reason is not resolved fully, the analysis will continue to fail.

4. Language for expressing analyses

An analysis result is undesirable if it is not what its client wants. For example, if the client is the constant propagation optimization it will want the underlying analysis to conclude that a variable's value is a constant. Therefore, non-constant values are undesirable for the constant propagation optimization. Undesirable results typically originate from one or more of the following sources:

- *Modeling of properties of the analyzed program*. For example, an analysis may assume that a call may modify any variable.
- *Data and control flow merges*. For example, a constant propagation may merge x=7 and x=5 to conclude that x is not a constant.
- *Underlying analyses*. For example, a constant propagation analysis may use the results of an imprecise call graph analysis.

If we wish to determine why an analysis produces undesirable results, we need to identify the above sources in the analysis code; we call these sources *failure points* since they may cause an analysis to fail. Once we have identified the failure points in an analysis, we can track exactly which failure points degrade each analysis result.

[3] For ease of explanation, we only consider boolean analyses for the time being with "true" being desirable and "false" being undesirable; later we show how we also handle non-boolean analyses.

Identifying failure points in analysis code is difficult (and in general impossible) if the analysis writer does not somehow mark these points in the code. Thus, for our work we write our analyses in a language that explicitly identifies failure points. Note that there are many alternative approaches for identifying failure points (e.g., adding annotations to the Java source code of the analysis or using naming conventions); our ideas on identifying reasons for analysis failure are applicable to all of these alternatives.

We now describe our language for writing analyses (Section 4.1), illustrate the language with an example (Section 4.2), and demonstrate that our language can express any data-flow analysis (Section 4.3).

4.1 Syntax and value semantics of the condition language

For ease of exposition, this section focuses only on analyses that produce boolean values. We consider "true" values to be desirable and "false" values to be undesirable. Section 6 describes how we handle analyses with arbitrary lattices.

Figure 1 gives the syntax of our language. Most constructs in our language (except for the *property* or simple variable reference) evaluate to a pair of a boolean (the value of the construct) and a reason of type $\mathcal{R}$. The value and the reason within the pair are related: If the value is true, the reason is "NoReason". If the value is false, the reason explains why the value is false. A variable reference's type usually depends on the type of some property (see below). Figure 2 describes how we compute the boolean values; Section 5 shows how we compute the reasons.

Our semantics (both for values and for reasons) use a pair of environments, (E, F), where environment E binds variables and F contains results of prior analyses. Both environments have two parts, one to store values and the other to store the reasons for those values. We subscript the environments with B (boolean values) or R (reasons) to indicate which part of the environment we mean.

We now describe the constructs of our language.

The *query* construct queries analyses results from user-defined analyses that have already completed or are in progress. Since two analyses may depend on each other, the computation of *query* may be cyclic. We use ideas from work on solving boolean equation systems [3, 7] to compute the fixed point of these cycles. Intuitively, our solution takes the dependence graph between multiple analyses and injects true or false (whichever is the neutral case) for queries involved in a cyclic dependency.

The *property* construct queries the results of analyses that are implemented externally to our system. The set of properties is not fixed; a user of our system can (and will) add new properties easily. Moreover, different properties may return entities of different types. For example an *allPaths* property returns a set of paths, an *allPredecessors* property returns a set of statements, and a *Modifies* property returns a boolean.

The first argument to both *query* and *property* identifies the query or property to look up. The remaining arguments are parameters to the query or property. For example, *query(isLive, s, x)* looks up whether or not x is live at statement s. The arguments to *query* and *property* may themselves be terms in our language; in this case we use our language's value semantics to evaluate the parameters.

forall iterates over the set of elements y (e.g., paths or statements) and evaluates z for each path or statement after binding x to the current element in the iteration. The *forall* evaluates to true only if z evaluates to true for all elements of the set. Thus, *forall* models a control merge. The $\wedge$ and $\vee$ model data merges.

The *unless* and *whenever* model decision points in the analysis.[4] The *unless* evaluates to true if the condition, x, evaluates to true;

```
⟨Term⟩ ::= ⟨Variable⟩
  | query(⟨Literal⟩, ··· )
  | property(⟨Literal⟩, ··· )
  | forall ⟨Variable⟩ : ⟨Term⟩ in ⟨Term⟩
  | unless (⟨Term⟩) then ⟨Term⟩
  | whenever (⟨Term⟩) ensure ⟨Term⟩
  | ⟨Term⟩ ∧ ⟨Term⟩
  | ⟨Term⟩ ∨ ⟨Term⟩
```

Figure 1: The syntax of the language for writing analyses

```
F(IsNotAReachingDef, d, u, x) ::=
    forall p: property(AllAcyclicPaths, d, u) in
        property(Modifies, property(ExcludeEndPoints, p), x)
```

Figure 3: Definition of IsNotAReachingDef

otherwise it evaluates to whatever y evaluates to. The *whenever* evaluates to true if both its condition, x, and its body, y, evaluate to true. Otherwise it evaluates to false.

4.2 Example

The following example shows how to write an *IsConstant0*, which determines whether or not variable x is a constant at statement s. Section 6 describes the machinery we need to write a more general *IsConstant*.

```
F(IsConstant0, s, x) ::=
    forall d: property(DefiningStatements, x) in
        unless query(IsNotAReachingDef, d, s, x) then
            property(RHSIs0, d)
```

IsConstant0 uses the following properties: *DefiningStatements* (returns the set of all potential definitions of a variable) and *RHSIs0* (whether or not the right-hand side of an assignment is 0). Intuitively, *IsConstant0* iterates over all assignments to x and if that assignment reaches s then *IsConstant0* checks that the right-hand side of the assignment is 0. *IsConstant0* also queries another analysis, *IsNotAReachingDef*, which returns true if a definition, d, of variable x does *not* reach use u.

We show the definition of *IsNotAReachingDef* in Figure 3. *IsNotAReachingDef* uses three properties: *AllAcyclicPaths* (returns the set of acyclic paths between two points), *Modifies* (determines if a path modifies a variable), and *ExcludeEndPoints* (which strips the end points out of a path). Intuitively, *IsNotAReachingDef* checks, for each acyclic path between d and u, whether or not the path modifies x.

On evaluating the above analyses on appropriate arguments, our system populates the environments, F_B and F_R, with the outcomes and reasons for the analyses.

Note that it is straightforward to mix parts written in other languages with parts written in our language by modeling them as properties.

4.3 Expressiveness

We now show how to express boolean data-flow analyses in our language. To simplify our argument, we present our argument using a forward data flow analysis. The argument for backward analyses is analogous. We assume that the data-flow analysis is in the form:

$$\begin{array}{lcl} A_{in}(s) & = & \bigsqcap\{A_{out}(s') | s' \rightarrow s\} \\ A_{out}(s) & = & f(A_{in}(s), s) \end{array}$$

Depending on whether the analysis is a meet or join analysis, $\bigsqcap$ is either $\bigwedge$ or $\bigvee$. For now we assume that $\bigsqcap$ is $\bigwedge$; later we discuss how we handle $\bigvee$. $s' \rightarrow s$ means that s is a direct successor of s' in

[4] Note that we use these constructs instead of the more usual *if*; Section 5 explains the reason for this choice.

$[\![\textit{query}(id,\cdots)]\!]_B(E,F) ::= F_B[\langle id,\cdots\rangle]$ $\quad$ $[\![x]\!]_B(E,F) ::= E_B[x]$

$[\![\textit{property}(id,\cdots)]\!]_B(E,F) := \textit{whatever the property computes}$ $\quad$ $[\![\textit{forall} x : y \textit{ in } z]\!]_B(E,F) ::= \forall\, a \in [\![y]\!]_B(E,F).[\![z]\!]_B(E_B[x\backslash a],F)$

$[\![x \wedge y]\!]_B(E,F) ::= [\![x]\!]_B(E,F) \wedge [\![y]\!]_B(E,F)$ $\quad$ $[\![x \vee y]\!]_B(E,F) ::= [\![x]\!]_B(E,F) \vee [\![y]\!]_B(E,F)$

$[\![\textit{unless } x \textit{ then } y]\!]_B(E,F) ::= [\![x \vee y]\!]_B(E,F)$ $\quad$ $[\![\textit{whenever } x \textit{ ensure } y]\!]_B(E,F) ::= [\![x \wedge y]\!]_B(E,F)$

Figure 2: The semantics $[\![]\!]_B$

the CFG of the program. We assume that $A_{in}(\textit{entry})$ is initialized according to the analysis.

To emulate this analysis in our language, we need to emulate A_{in} and A_{out}. To emulate A_{in} we use the property *Predecessors* which returns the immediate predecessors of its argument statement.

$$A_{in}(s) = \textit{forall}\, x : \textit{property}(\texttt{Predecessors}, \text{s})\, \textit{in}\, \textit{query}(A_{out}, x)$$

To emulate A_{out} we need to emulate f. To express f in our language, we rewrite $f(x,s)$ to be $(x \wedge \neg f_1(s)) \vee f_2(s)$ (any boolean function can be expressed in this form). Many data flow formulations (e.g., using Kill and Gen sets) already express f in this form. We can use properties to implement $\neg f_1$ and f_2.

To emulate $\bigvee$ (i.e., join) analyses in our system, the most obvious approach is to frame join analyses as meet analyses (we use duals of the properties to do this) and then use the above approach.

In summary, we can straightforwardly express boolean dataflow analyses in our language.

5. Computing reasons for boolean analyses

There are two main insights that enable us to derive reasons for analysis failure.

First, as already discussed in Section 4, we explicitly mark all failure points in analyses; this enables us to track exactly the points that contribute to analysis failure.

Second, we express analyses using positive logic only; the only language construct that can produce false on its own is the *property* construct. This enables us to easily identify those invocations of property that contribute to undesirable results. For example, compare the definition of IsNotAReachingDef from Figure 3, written in positive logic, to the following analysis, written in full logic (which uses the *exists* operator which our language does not include):

F(`IsAReachingDef`, d, u, x) ::=
exists p: *property*(`AllAcyclicPaths`, d, u) *in*
property(`DoesNotModify`, *property*(*ExcludeEndPoints*, p), x)

If the first analysis evaluates to false, it must be the case that one or more of the *property(Modifies, p, x)* evaluated to false. The properties that evaluated to false are exactly the properties that contribute to the reason for the undesirable result of the whole analysis. The user must address all of the false properties before the analysis can succeed.

If the second analysis evaluates to false, then it must be the case that there is no path for which *property(DoesNotModify, p, x)* is true. The user can address *any one* of the failing *property(DoesNotModify, p, x)* before the analysis succeeds.

In our experience using our system, we have found the first case to be more manageable than the second. The reason for this is that the first case enumerates exactly the reasons that the user has to address in order to make the analysis succeed (and no more or no less); in the second case the user can address one of many reasons but the user does not have any further guidance on which one to actually try to address. Thus we do not allow "not" or "exists" in our language. We could modify our system to compute reasons even without these restrictions or use properties to get around these restrictions; it is just that we have found that our reasons are more useful to the user with these restrictions.

Listing 1. Example

```
x = 1;
if (···) x = 2;
else a = 0;
a = 7;
a = x;
```

5.1 Semantics for computing reasons

The reason semantics (Figure 4) for our language specify how to compute reasons for undesirable analysis results. To compute the reasons, the reason semantics make use of the value semantics (Figure 2). Specifically, the reason semantics produce a reason only for constructs that evaluate to false (and thus contribute to undesirable analysis results). Note how disallowing "not" and "exists" in our language also simplifies these semantics; without these restrictions we also have to compute reasons for constructs that evaluate to "true" in case the "true" construct is surrounded by a "not".

To obtain the reason for a *query* or a variable reference, we simply look up the reason in the appropriate environment. Note that we may have to evaluate the arguments using the value semantics before we can perform the lookup.

If a boolean *property* evaluates to true, then we do not need a reason (recall that we track reasons only for undesirable results). If a boolean *property* evaluates to false, we label the reason as "not" of the property. This is the base case for constructing a reason.

If $\wedge$ evaluates to true, we do not need a reason. If only one of x or y is false, the reason is the reason for the false one. If both x and y are false, then the reason is the AndReason of the two reasons. If $\vee$ evaluates to true, we do not need a reason. If it evaluates to false, the reason is the OrReason of the reasons for x and y.

For *unless* and *whenever* we use the rules for $\vee$ and $\wedge$ to generate reasons.

If a *forall* evaluates to true, then we do not need a reason. If it evaluates to false, then all instances of z that evaluated to false must be addressed before the *forall* succeeds. Thus, we construct an AndReason with the reasons of all the false instances of z.

5.2 Example

Let's suppose we wish to apply the *IsNotAReachingDef* analysis from Figure 3 to the code in Listing 1. If we instantiate the analysis as *IsNotAReachingDef*($x = 1, a = x, x$), we find that the analysis fails. From the *forall* rule in Figure 4 we compute the reason AnalysisFailed(not-Modifies, a=0 $\rightarrow$ a=7, x); in other words, *IsNotAReachingDef* failed because along one path (passing through $a = 0$ and $a = 7$) the definition actually reaches $a = x$

6. Computing reasons for non-boolean analyses

Many program analyses compute non-boolean values. For example, a constant propagation analysis determines whether or not a variable reference is a constant as well as the value of the constant.

$$[\![\textit{query}(id, \cdots)]\!]_{\mathcal{R}}(E, F) ::= F_{\mathcal{R}}[\langle id, \cdots \rangle]$$

$$[\![x]\!]_{\mathcal{R}}(E, F) ::= E_{\mathcal{R}}[x]$$

$$[\![property(id, x, y)]\!]_{\mathcal{R}}(E, F) := \begin{cases} \textit{NoReason} & | \quad [\![\textit{property}(id, x, y)]\!]_B(E, F) = \textit{true} \\ \textit{AnalysisFailed(not-id, } x, y\textit{)} & | \quad [\![\textit{property}(id, x, y)]\!]_B(E, F) = \textit{false} \end{cases} \qquad \textit{Only for boolean properties}$$

$$[\![\textit{unless } x \textit{ then } y]\!]_{\mathcal{R}}(E, F) ::= [\![x \vee y]\!]_{\mathcal{R}}(E, F)$$

$$[\![\textit{whenever } x \textit{ ensure } y]\!]_{\mathcal{R}}(E, F) ::= [\![x \wedge y]\!]_{\mathcal{R}}(E, F)$$

$$[\![\textit{forall } x : y \textit{ in } z]\!]_{\mathcal{R}}(E, F) ::= \begin{cases} \textit{NoReason} & | \quad [\![\textit{forall } x : y \textit{ in } z]\!]_B(E, F) = \textit{true} \\ \textit{AndReason}(\quad \{[\![z]\!]_{\mathcal{R}}(E[x \backslash a], F) & \\ \qquad |a \in [\![y]\!]_B(E, F) \wedge [\![z]\!]_B(E[x \backslash a], F) = \textit{false}\}) & | \quad [\![\textit{forall } x : y \textit{ in } z]\!]_B(E, F) \neq \textit{true} \end{cases}$$

$$[\![x \wedge y]\!]_{\mathcal{R}}(E, F) ::= \begin{cases} \textit{NoReason} & | \quad [\![x]\!]_B(E, F) = \textit{true} \wedge [\![y]\!]_B(E, F) = \textit{true} \\ [\![x]\!]_{\mathcal{R}}(E, F) & | \quad [\![x]\!]_B(E, F) = \textit{false} \wedge [\![y]\!]_B(E, F) = \textit{true} \\ [\![y]\!]_{\mathcal{R}}(E, F) & | \quad [\![x]\!]_B(E, F) = \textit{true} \wedge [\![y]\!]_B(E, F) = \textit{false} \\ \textit{AndReason}([\![x]\!]_{\mathcal{R}}(E, F), [\![y]\!]_{\mathcal{R}}(E, F)) & | \quad [\![x]\!]_B(E, F) = \textit{false} \wedge [\![y]\!]_B(E, F) = \textit{false} \end{cases}$$

$$[\![x \vee y]\!]_{\mathcal{R}}(E, F) ::= \begin{cases} \textit{NoReason} & | \quad [\![x]\!]_B(E, F) = \textit{true} \vee [\![y]\!]_B(E, F) = \textit{true} \\ \textit{OrReason}([\![x]\!]_{\mathcal{R}}(E, F), [\![y]\!]_{\mathcal{R}}(E, F)) & | \quad [\![x]\!]_B(E, F) = \textit{false} \wedge [\![y]\!]_B(E, F) = \textit{false} \end{cases}$$

Figure 4: The semantics $[\![]\!]_{\mathcal{R}}$

To support these analyses, we generalize our language constructs to compute a value from an analysis's lattice, $\mathcal{L}$. We first describe how we compute reasons for non-boolean analyses. Then we give an example that demonstrates our approach.

6.1 The $\mathcal{L}$ and $\mathcal{R}_{\mathcal{L}}$ semantics

Figure 5 defines how we produce the result from $\mathcal{L}$ for a meet analysis (the rules for a join analysis are similar and and thus omitted from paper). Note that the E and the F environments now have a $\mathcal{L}$ part instead of a B part.

The semantics in Figure 5 use $\sqcup_{\mathcal{L}}$ and $\sqcap_{\mathcal{L}}$ which the analysis writer provides to our system (i.e., they are not written in our language); they implement join and meet on the analysis's lattice.

For boolean analyses we had defined "true" as a desirable value and "false" as undesirable. For non-boolean analyses, the user of our system provides a function *desirable*$_{\mathcal{L}}$ which takes values of the analysis's lattice $\mathcal{L}$ and returns true or false; in essence, the user provides a way for our system to determine whether or not a given analysis result is desirable. The user provides one implementation of *desirable*$_{\mathcal{L}}$ for each analysis's lattice. When we write just *desirable* (i.e., omit the subscript $\mathcal{L}$) we intend it as an overloaded operator; our system picks the appropriate implementation of *desirable* based on the domain of the argument. We assume that top of $\mathcal{L}$ ($\top_{\mathcal{L}}$) is desirable, and $\bot_{\mathcal{L}}$ is undesirable.

For each use of a property, *desirable*$_{\mathcal{L}}$ effectively distinguishes between situations that allow the transformation (desirable) and situations that disallow the transformations (undesirable). A property (or query) may produce undesirable results in two ways: (i) the property or query is actually provably undesirable; or (ii) the property or query uses an imprecise analysis and that causes it to produce undesirable results.

In our system, desirable values are always maximally precise: As every desirable value is *provably* desirable, no amount of added precisions in our analyses will ever make a desirable result undesirable. In contrast, undesirable values are always maximally imprecise: A better analysis could easily make an undesirable value desirable.

To make our system run fast without sacrificing precision, it transparently swaps analyses with more precise ones during the evaluation of a client analysis, if the client analysis produces too imprecise results. To allow such a transparent swap, all clients of the analysis must – when given the choice – use the more precise values of the new analysis instead of the imprecise values of the old analysis. If implemented navely, this approach complicates client analysis since before they can use a value, they need to confirm that the value is not superseded by a more precise value.

We sidestep this complication by using a single set of operators for comparing both precision and desirability of values: Since added precision may only make a value become desirable, and not undesirable, every comparison of desirability will always examine precision, too. This eliminates the burden of having to do a second comparison, and allows our system to replace an analysis with a more precise analysis without any additional effort on part of the client analysis.

The semantics in Figure 5 are straightforward: they use the user-provided meet or join operator instead of using boolean $\vee$ and $\wedge$ when combining values. The x for a *whenever* or *unless* must evaluate to a boolean; it is up to the user ensure this (e.g., by using the same criteria as desirable). When the x of an *unless* evaluates to true, the whole *unless* succeeds and produces the top value of the lattice; the reason for this is that we can meet any value with the top without degrading analysis results. When the x value of a *whenever* evaluates to false, the whole *whenever* fails; thus it produces the bottom value of the lattice to indicate analysis failure.

Figure 6 gives the semantics for computing reasons when using the $\mathcal{L}$ lattice and assuming a meet analysis. The semantics for a join analysis are analogous and so we omit them.

For a *query* or a variable reference, we simply look up the reason in the appropriate environment.

For a *property* we evaluate the property using the value semantics and check if the value is desirable; if it is desirable we do not need a reason. If the value is not desirable, we produce a reason that is the "not-" of the property.

For an *unless* we do not need a reason if the *unless* produces a desirable result. If it produces an undesirable result then we produce an OrReason since addressing either x or y will cause the *unless* to produce a desirable result.

For a *whenever* we do not need a reason if the whenever produces a desirable result. If either x is false or y is undesirable, we return the reason of the undesirable one. If both x and y are undesirable, we return the AndReason of both reasons.

For a *forall* we do not need a reason if the *forall* produces a desirable result. If it produces an undesirable result, it could be due to one of two reasons: (i) z produced an undesirable result for one or more of the bindings of y; (ii) z produced a desirable result for all bindings but the meet of the results was undesirable. In the first case, we produce an *AndReason* of all the instances of z that produced undesirable results; in the second case we produce a *MeetFailed Reason*.

$$\llbracket \mathit{query}(id, \cdots) \rrbracket_{\mathcal{L}}(E, F) ::= F_{\mathcal{L}}[\langle id, \cdots \rangle] \llbracket x \rrbracket_{\mathcal{L}}(E, F) ::= E_{\mathcal{L}}[x]$$

$$\llbracket \mathit{property}(id, \cdots) \rrbracket_{L}(E, F) := \textit{whatever the property computes} \qquad \llbracket \textit{forall } x : y \textit{ in } z \rrbracket_{\mathcal{L}}(E, F) ::= \bigsqcap a \in \llbracket y \rrbracket_{\mathcal{L}}(E, F). \llbracket z \rrbracket_{\mathcal{L}}(E_{\mathcal{L}}[x \backslash a], F)$$

$$\llbracket x \vee y \rrbracket_{\mathcal{L}}(E, F) ::= \llbracket x \rrbracket_{\mathcal{L}}(E, F) \sqcup_{\mathcal{L}} \llbracket y \rrbracket_{\mathcal{L}}(E, F) \qquad \llbracket x \wedge y \rrbracket_{\mathcal{L}}(E, F) ::= \llbracket x \rrbracket_{\mathcal{L}}(E, F) \sqcap_{\mathcal{L}} \llbracket y \rrbracket_{\mathcal{L}}(E, F)$$

$$\llbracket \textit{unless } x \textit{ then } y \rrbracket_{L}(E, F) ::= \begin{cases} \top_{\mathcal{L}} & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{true} \\ \llbracket y \rrbracket_{\mathcal{L}} & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{false} \end{cases} \qquad \llbracket \textit{whenever } x \textit{ ensure } y \rrbracket_{L}(E, F) ::= \begin{cases} \llbracket y \rrbracket_{\mathcal{L}} & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{true} \\ \bot_{\mathcal{L}} & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{false} \end{cases}$$

Figure 5: The semantics $\llbracket\rrbracket_{\mathcal{L}}$ (assuming an all-paths analysis)

$$\llbracket \mathit{query}(id, \cdots) \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) ::= F_{\mathcal{R}}[\langle id, \cdots \rangle]$$

$$\llbracket x \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) ::= E_{\mathcal{R}}[x]$$

$$\llbracket property(id, x, y) \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) := \begin{cases} \textit{NoReason} & | \quad \textit{desirable}(\llbracket \textit{property}(id, x, y) \rrbracket_{\mathcal{L}}(E, F)) = \textit{true} \\ \textit{AnalysisFailed(not-id, } x, y) & | \quad \textit{desirable}(\llbracket \textit{property}(id, x, y) \rrbracket_{\mathcal{L}}(E, F)) = \textit{false} \end{cases}$$

$$\llbracket \textit{unless } x \textit{ then } y \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) ::= \begin{cases} \textit{NoReason} & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{true} \\ \textit{NoReason} & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{false} \wedge \textit{desirable}(\llbracket y \rrbracket_{\mathcal{L}}(E, F)) = \textit{true} \\ \textit{OrReason}(\llbracket x \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F), \llbracket y \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F)) & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{false} \wedge \textit{desirable}(\llbracket y \rrbracket_{\mathcal{L}}(E, F)) = \textit{false} \end{cases}$$

$$\llbracket \textit{whenever } x \textit{ ensure } y \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) ::= \begin{cases} \textit{NoReason} & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{true} \wedge \textit{desirable}(\llbracket y \rrbracket_{\mathcal{L}}(E, F)) = \textit{true} \\ \llbracket y \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{true} \wedge \textit{desirable}(\llbracket y \rrbracket_{\mathcal{L}}(E, F)) = \textit{false} \\ \llbracket x \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F) = \textit{false} \wedge \textit{desirable}(\llbracket y \rrbracket_{\mathcal{L}}(E, F)) = \textit{true} \\ \textit{AndReason}(\llbracket x \rrbracket_{\mathcal{R}}(E, F), \llbracket y \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) & | \quad \llbracket x \rrbracket_{\mathcal{L}}(E, F)) = \textit{false} \wedge \textit{desirable}(\llbracket y \rrbracket_{\mathcal{L}}(E, F)) = \textit{false} \end{cases}$$

$$\llbracket \textit{forall } x : y \textit{ in } z \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) ::= \begin{cases} \textit{AndReason}(\{\llbracket z \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E[x \backslash a], F) | a \in \llbracket y \rrbracket_{\mathcal{L}}(E, F) \wedge & | \quad \exists a \in \llbracket y \rrbracket_{\mathcal{L}}(E, F). \neg \textit{desirable}(\llbracket z \rrbracket_{\mathcal{L}}(E[x \backslash a], F)) \\ \qquad \neg \textit{desirable}(\llbracket z \rrbracket_{\mathcal{L}}(E[x \backslash a], F)))\}) & \\ \textit{MeetFailed Reason}(\{a | a \in \llbracket y \rrbracket_{\mathcal{L}}(E, F)\}) & | \quad \forall a \in \llbracket y \rrbracket_{\mathcal{L}}(E, F). \textit{desirable}(\llbracket z \rrbracket_{\mathcal{L}}(E[x \backslash a], F)) \wedge \\ & \quad \neg \textit{desirable}(\bigsqcap_{a \in \llbracket y \rrbracket_{\mathcal{L}}(E, F)} \llbracket z \rrbracket_{\mathcal{L}}(E[x \backslash a], F)) \\ \textit{NoReason} & | \quad \textit{otherwise} \end{cases}$$

$$\llbracket x \wedge y \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) ::= \begin{cases} \textit{NoReason} & | \quad \textit{desirable}(\llbracket x \sqcap_{\mathcal{L}} y \rrbracket_{\mathcal{L}}(E, F)) \\ \llbracket x \rrbracket_{\mathcal{R}_{\mathcal{L}}} & | \quad \neg \textit{desirable}(\llbracket x \rrbracket_{\mathcal{L}}(E, F)) \wedge \textit{desirable}(\llbracket y \rrbracket_{\mathcal{L}}(E, F)) \\ \llbracket y \rrbracket_{\mathcal{R}_{\mathcal{L}}} & | \quad \textit{desirable}(\llbracket x \rrbracket_{\mathcal{L}}(E, F)) \wedge \neg \textit{desirable}(\llbracket y \rrbracket_{\mathcal{L}}(E, F)) \\ \textit{MeetFailed Reason}(x, y) & | \quad \textit{otherwise (i.e.}\neg \textit{desirable}(\llbracket x \rrbracket_{\mathcal{L}}(E, F) \sqcap \llbracket y \rrbracket_{\mathcal{L}}(E, F))) \end{cases}$$

$$\llbracket x \vee y \rrbracket_{\mathcal{R}_{\mathcal{L}}}(E, F) ::= \begin{cases} \textit{NoReason} & | \quad \textit{desirable}(\llbracket x \sqcup_{\mathcal{L}} y \rrbracket_{\mathcal{L}}(E, F)) \\ \textit{OrReason}(\llbracket x \rrbracket_{\mathcal{R}_{\mathcal{L}}}, \llbracket y \rrbracket_{\mathcal{R}_{\mathcal{L}}}) & | \quad \neg \textit{desirable}(\llbracket x \rrbracket_{\mathcal{L}}(E, F)) \textit{ and } \neg \textit{desirable}(\llbracket y \rrbracket_{\mathcal{L}}(E, F)) \\ \textit{JoinFailed Reason}(x, y) & | \quad \textit{otherwise (i.e.}\neg \textit{desirable}(\llbracket x \rrbracket_{\mathcal{L}}(E, F) \sqcup \llbracket y \rrbracket_{\mathcal{L}}(E, F))) \end{cases}$$

Figure 6: The semantics $\llbracket\rrbracket_{\mathcal{R}_{\mathcal{L}}}$ (assuming an all-paths analysis)

For an $\vee$ ($\wedge$), if both (one) of the arguments are undesirable, we use the reason from the undesirable arguments. If both the arguments are desirable but the outcome of the $\vee/\wedge$ is undesirable, then it produces a *JoinFailed Reason*/*MeetFailed Reason*.

Our reason semantics and discussion of the *forall*, $\vee$, and $\wedge$ are simplified for the purpose of presentation. To see this, let's suppose we are writing a constant propagation analysis and we do $7 \wedge \bot$ (this may arise because a variable has value 7 on one path and a value $\bot$ (i.e., not a constant) on another path). Our reason semantics will use the reason for the $\bot$ as the reason for the undesirable outcome of the $\wedge$. While this is part of the reason, it is not a sufficient reason: the variable must not only have a non-$\bot$ value on both paths but the two values must be the same. Our system actually handles this by producing a hierarchical reason: the top-level reason is *MeetFailed Reason*($7, \bot$); below it there is a reason for the $\bot$.

6.2 Example

Given a *reaching definitions* analysis, writing a simple constant propagation is easy. The following produces the constant value of x at statement s if x is a constant, and $\bot$ otherwise. The analysis writer provides a meet operator, which returns its argument if both arguments are the same, the non-$\top$ argument if one of the arguments is $\top$, and $\bot$ otherwise (i.e., a standard constant propagation lattice). The analysis writer also implements *desirable*, which returns false if its argument is $\bot$ and true otherwise.

```
F(ConstantPropagation, s, x) ::=
  forall d: query(ReachingDefinitions, s, x) in
    property(getRhs, s)
```

If the different reaching definitions return different constant values (say 7 and 5), our system produces a *MeetFailed Reason(7,5)* which tells the user that constant propagation produced an undesirable result because 7 and 5 are different. The user can then explore why the analysis considered the "7" value; the reason computation for "ReachingDefinitions" produces this reason.

7. Properties of Reasons

Recall from Section 3 that we want reasons to be necessary and sufficient. These properties are desirable since they guarantee that a user of our approach will (i) not waste time addressing issues that do not need to be addressed (necessary); and (ii) not find that an analysis produces undesirable results even after the user has addressed all the reasons (sufficient).

The full proofs that our reasons are necessary and sufficient uses structural induction over the language presented in Figures 1. Since our proofs are straightforward, we only sketch them here and only for the boolean analyses. The proof for the non-boolean analyses, over the hierarchical reasons of our full system, is similar.

To see why our reasons are necessary, we first consider conditions that do not use "$\vee$" or *unless*. From Figure 4 we see that only properties that evaluate to false end up as part of the reason (either directly or embedded inside of *AndReason*). As, by construction, these properties must be satisfied in order for the analysis to evaluate to true, our system produces necessary reasons. For analyses that use "$\vee$" we construct an *OrReason*; we trivially see that satisfying the *OrReason* will satisfy the "$\vee$" condition and thus our reasons are necessary in the presence of "$\vee$". For conditions that use *unless* we generate an *OrReason* when both the condition and body of the *unless* evaluate to false. In this case, we can satisfy the *unless* by either making its condition or its body evaluate to true; this is exactly what the *OrReason* for this case of the *unless* captures. Thus, our reasons are necessary for our full language.

The key insight behind our proof that our reasons are sufficient is that our analyses only use positive logic and thus the analysis fails only as a result of one or more terms that evaluate to false. From the semantics in Figure 4 we see that our reasons incorporate all terms that cause an analysis to fail. Specifically, our reasons omit all true terms and those false terms whose falsehood does not propagate up to the analysis (e.g., false terms that are part of a disjunct with the other term in the disjunct evaluating to true). Thus, our reasons are sufficient. This property is much harder to prove if our analysis can use negative logic.

In summary, we have shown that our approach produces reasons that are necessary and sufficient for an analysis's failure.

8. Using our system

We envision two kinds of uses of our approach: in program transformation tools and in program understanding tools.

In program transformation tools, each transformation has an applicability condition that describes whether or not a given application of the transformation is legal (i.e., preserves semantics). If a user requests an illegal transformation, the user may wish to know why the transformation is illegal. Our system naturally produces reason for this illegality for boolean and non-boolean analyses.

In a program understanding tool, our system may not immediately produce the reason that the user needs. In particular, for lattices whose elements are sets, our system computes a single reason instead of one reason for each element within the set

As an example of this, suppose a programmer requests a program slice and the slice is much larger than what the programmer expected. How should the user define *desirable*? If *desirable* returns a true if a slice is small (perhaps using some arbitrary threshold) and otherwise returns false, our system produces a reason for large slices. This reason describes why the slice is large. This may not be what the user wants. Often, the user will want to know why a particular statement *is* or *is not* in the slice.

We can get such reasons by defining *desirable* so that it checks just for the statement of interest. While this approach works, it is cumbersome: we would effectively have to rerun the reason computation once for each such query by the user; specifically, once the user knows why s_1 is in the slice, she may wish to know why s_2 is in the slice, and so on.

In other words, for analyses whose lattice elements are sets, we may wish to associate a reason with each element in the set rather than a single reason for the entire set. In terms of our slicing example, rather than having a single reason for why the slice is large or why a particular statement is in the slice, we can have a reason pre-computed for each statement that belongs to the slice. Our system automatically does this when the user-provided meet and join operators are set union and intersection (or vice versa).

To see how this works, let's suppose that the reaching definition needs to meet definitions reaching along two paths; along one path the definitions are d_1 and d_2 and along the other path the definitions are d_2 and d_3. Furthermore, assume we have reasons r_a and r_b for d_1 and d_2 along the first path and reasons r_c and r_d for definitions d_2, and d_3 on the second path. Since d_2 arrives from both paths, the reason for it after the meet is *AndReason(r_b, r_c)*. As d_1 and d_3 occur only on one path, their reasons remain r_a and r_d.

Listing 2. Analysis for LICM

```
1 F(LICM, toRemove) ::=
2  property(cannotModify, toRemove, property(RHS, toRemove))
3  ∧
4  forall path: property(ExcludeEndPoints,
5                 property(AllAcyclicPaths, toRemove, toRemove)) in
6    forall stmt: property(allStatementsInPath, path) in
7      (property(CannotModify, stmt, property(LHS, toRemove))
8       ∧
9       property(CannotModify, stmt, property(RHS, toRemove))
10      ∧
11      unless property(CannotThrow, toRemove) then
12        (property(cannotModifyAnyState, stmt)
13         ∧
14         property(CannotThrow, stmt)))
```

Listing 3. Program with an opportunity for LICM

```
boolean isEqual(Iterator<Integer> iter) {
  while (b) {
    int v = x.f;
    int v2 =  iter.next();
    if (v != v2) { return false; }
  }
  return true;
}
```

9. An example

In order to demonstrate how one goes about using our system, we now present a full example which implements the analysis for loop-invariant code motion (LICM). The analysis (Listing 2) evaluates to true if *toRemove* can be hoisted outside its enclosing loop (for simplicity our condition assumes that there is exactly one loop that surrounds *toRemove*).

Line 2 ensures that executing `toRemove` does not change its own subsequent behavior. The two *forall* iterate over all paths through the loop and all statements in those paths. Lines 7 and 9 make sure that no statement modifies the left or right-hand side of *toRemove*. The *unless* in line 11 ensures that either *toRemove* does not throw an exception or if it does, then other statements in the loop do not throw exceptions or modify any variables. This ensures that LICM respects Java's exception semantics.

An implementation of LICM that handles all cases (such as jumps out of the loop) is 139 lines in our condition language. The analysis in Listing 2 ignores many corner cases for ease of exposition.

If we evaluate our LICM condition on the code in Listing 3 with `v=x.f` as *toRemove* we get the reason in Figure 7. The top-level *AndReason* in Line 2 says that the user must address both of its children (Lines 3 and 6) to enable LICM.

The sub-reason in Line 3 comes directly from Lines 7–9 in Listing 2. Specifically, the reason says that the call to `next` may modify either the left or the right-hand side of *toRemove*. The sub-reason in Line 6 comes directly from Lines 11–14 in Listing 2. Specifically, it says that moving *toRemove* out of the loop may violate Java's exception semantics either by reordering exceptions (in case `x.f` or the call to `next` throws an exception) or by reordering exceptions with other side effects (since the call to `next` may modify any state).

```
Optimization "LICM" for statement "int v = x.f"
AndReason:
  AndReason:
    AnalysisFailed(CanModify, "v2 = iter.next()", "x.f")
    AnalysisFailed(CanModify, "v2 = iter.next()", "v")
  OrReason:
    AnalysisFailed(CanThrowException, "int v = x.f")
    AndReason:
      AnalysisFailed(CanModifyAnyState, "v2 = iter.next()")
      AnalysisFailed(CanThrowException, "int v2 = iter.next()")
```

Figure 7: A reason for the failure of LICM's condition

10. Evaluation

We have already proved that our system produces reasons that are necessary and sufficient and that our analysis language can express arbitrary data flow analyses. This section experimentally evaluates our approach.

10.1 Methodology

To evaluate our system for computing reasons, we embedded our system in our interactive optimizer. Our interactive optimizer implements interprocedural optimizations for the Java programming language using the approach outlined in this paper. As a result, our system alerts a user whenever an optimization has only a few failure reasons (i.e., *almost* applies). Our system then interacts with the user by providing a reason for the inapplicability of the optimization. The user responds by possibly adding assumptions about the program which enable the optimizer to apply many of the almost applicable optimizations.

We now discuss the implementation of a few traditional compiler optimizations (specifically constant propagation, copy propagation, dead assignment elimination, and loop invariant code motion) in our system. We limit ourselves to simple, well-known optimizations to make the discussion more accessible to the reader.

Our optimizer breaks down the task of determining the applicability of an optimization into two parts. First, the optimizer uses simple syntactic pattern matching to identify possible optimization opportunities. For example, a simple pattern may indicate that a use of a variable may be replaced by a constant if there is a preceding assignment of a constant value to the variable. Second, the optimizer uses program analyses (implemented in the language in Section 4) to check whether the optimization is actually applicable. For example, our syntactic pattern matching cannot account for aliases; the second stage incorporates aliasing information to check the legality of the constant propagation. If the second stage finds that the optimization is not really applicable, it produces reasons which the user can address by providing assumptions.

We used this system to optimize all applications taken from SPECjvm98 [12] benchmark suite that are available as Java source code and SPECjbb [11] benchmark. We conducted our experiments on a 3.0GHz CoreDuo workstation with 8GB of memory.

Table 1 summarizes our results. The *Size(LOC)* row gives the size of the benchmarks. These sizes include the size of any libraries that are shipped with the benchmarks. Since our optimizations are interprocedural, they also analyze these libraries along with the application code.

The remainder of the table has one section for each optimization. In addition, it has a section for "Reaching Def": for this analysis, our system computes the reaching definitions of all variables used in all statements and produces a reason for each statement that ends up in a reaching definitions set.

The *Applicable* rows for the optimizations give the number of times the optimization was applicable without needing any input from the user. The *Inapplicable* rows give the number of times the optimization was inapplicable and thus we had to produce reasons

	Jess	SpecJBB	DB	compress	raytrace
Size (LOC)	22215	30777	11601	11370	14183
Constant Propagation (85 lines)					
Applicable	14662	19071	8899	9070	10648
Inapplicable	28975	38204	13766	13949	17912
Time (sec)	7.81	11.34	3.64	3.82	4.21
Copy Propagation (98 lines)					
Applicable	37339	49676	20213	20482	25119
Inapplicable	7690	10585	3329	3222	4166
Time (sec)	94.47	51.08	17.49	15.79	19.52
Dead Assignment Elimination (11 lines)					
Applicable	1409	650	13	5	7
Inapplicable	13113	19589	7872	8026	9285
Time (sec)	3.88	4.15	2.06	1.08	1.42
LICM (139 lines)					
Applicable	1140	1374	512	491	591
Inapplicable	169	149	96	86	91
Time	1.01	1.31	0.51	0.41	0.48
Reaching definitions (93 lines)					
Evaluations	30409	38376	15114	15353	19760
Time (sec)	21.75	23.23	6.17	22.19	22.54

Table 1. Summary statistics

for the failure of the optimization. The *Time* rows give the time (in seconds) to compute the reasons for *all* the inapplicable cases. The *Evaluations* row (for reaching definitions only) gives the number of different reaching definitions sets we computed (there is one for each use of a variable).

From these results we see that the time to compute reasons depends on the number of reasons we compute (which depends on the number of times an optimization is inapplicable). If we compute only a few reasons (e.g., 169 for LICM on *Jess*), our system takes only about a second; if we compute many reasons (e.g., 37339 for copy propagation on *Jess*) our system takes longer (94.47 seconds). Our system takes on average 0.1 seconds to produce a single reason and thus our system is fast enough for interactive use.

10.2 Difficulty of writing analyses

If our approach to makes analyses too hard to write, it will not be practical. To explore this issue, we wrote a number of analyses in our language (Section 10.1). Table 1 gives the number of lines of code we had to write for each analysis in our language. We see the analyses are quite small (ranging from 11 lines to 139 lines) especially considering that these analyses are interprocedural and handle the full Java language. The analyses were straightforward to write. In fact, we found the reasons extremely helpful for debugging the analyses since they told us exactly why our analyses were producing poor results; we used these reasons to selectively tune our analyses so that the analyses enabled the most optimization opportunities.

10.3 Size of reasons

The reasons computed by PT are small: over 96% of reasons for our compiler optimizations consist of less than 3 root reasons. Computing these reasons is fast: Over 95% of the reasons PT computes are computed within 0.1 seconds or less.

Figure 8 gives an example of a small reason for *Jess* along with the relevant code fragment from *Jess*. Note that we translate our reasons into prose for the user's convenience. This is a reason of size 1 which says that constant propagation failed because `initialStatus` could have one of two values.

Figure 9 gives an example of a large reason for *Jess* (this time we do not give the relevant code for *Jess* because it is too large to

```
  String initialStatus = "Ready";
  if (!testMode) {
     [...]
      initialStatus = "Error";
  }
  [...]
  showStatus(initialStatus);
================================================================
Optimization "ConstantProp" in Line 779:
MeetFailed, Two values of "initialStatus" were inequal:
  -> initialStatus = "Ready", line 747
  -> initialStatus = "Error", line 760
```

Figure 8: A typically small reason for constant propagation in Jess

```
Optimization "ConstantProp" in Line 585:
And:
  Inequal:
    -> tok = null, line 453
    -> non-constant value due to propagation of undesirable.
  PropagationOfUndesirable:
    The result of nextToken() may not be constant, line 455
  PropagationOfUndesirable:
    The result of nextToken() may not be constant, line 487
  PropagationOfUndesirable:
    The result of nextToken() may not be constant, line 500
  PropagationOfUndesirable:
    The result of nextToken() may not be constant, line 515
  PropagationOfUndesirable:
    The result of nextToken() may not be constant, line 533
  PropagationOfUndesirable:
    The result of nextToken() may not be constant, line 540
  PropagationOfUndesirable:
    The result of nextToken() may not be constant, line 560
```

Figure 9: An atypically large reason for constant propagation in Jess

reproduce here). This reason says that constant propagation failed because one of the reaching definitions for `tok` was `null` and the others were not constant (i.e., $\perp$). Our system tags reasons with *PropagationOfUndesirable* in its handling of the *forall* construct; specifically, it marks reasons as *PropagationOfUndesirable* when the values being merged along the different paths are already undesirable to start with.

In summary, we see that most of the reasons that our system computes are small and thus will not overwhelm users.

10.4 Usefulness of reasons

We have shown that our system produces reasons that are usually small and it produces them quickly; however, are the reasons actually useful? To determine this, we examined many of the reasons produced for the optimizations in Table 1 and tried to address the reasons. We addressed reasons by providing assumptions to our system. After providing the assumptions, we ran the programs and compared their output to the unoptimized program; in this way we confirmed, experimentally, that our optimizations were correct (and thus our reasons were correct). We now report our experience in providing assumptions for the copy propagation optimization.

On examining the reasons for all three benchmarks, we found that four reasons recurred in all benchmarks; we describe them here. Table 2 shows the number of copy propagation opportunities that our assumptions enabled. We now describe the assumptions and the reasons that led to them.

Assumption 1 tells our optimizer to disregard dynamic class loading. Assumption 2 tells our optimizer to disregard the possibility of `NullPointerException`. Our reasons indicated that these two factors frequently inhibit copy propagation. Since we knew that these benchmarks do not rely on dynamic class loading or on catching `NullPointerExceptions` we added the two assumptions.

Assumption 3 tells our optimizer to assume that library methods cannot modify instance variables of the application's classes. Many

Assumption	Jess	SpecJBB	DB	compress	raytrace
#1 and #2	25	29	17	57	119
#3	546	505	377	368	394
#4	1033	1188	639	20	11

Table 2. Addressing reasons for copy propagation

failure reasons referred to this possibility; we knew that to be false and thus added the assumption.

Assumption 4 tells our optimizer to assume that library methods will not call the application's methods. Once again, many failure reasons referred to this possibility.

Note that these assumptions are "macro" assumptions: i.e., rather than saying that "method f in library cannot modify instance variable i in class C", we simply said "no method in library can modify an instance variable in an application class". Our system uses reason abstraction: it combines similar concrete reasons to more abstract reasons and thus enables the user to address many reasons with a single assumption.

In summary, we found our reasons to be useful: we were able to address many reasons by adding a handful of assumptions. Adding these assumptions enabled many optimization opportunities and thus the reasons were useful.

11. Related Work

We are not aware of any prior work on a general approach for producing reasons for analysis failure. Interactive transformations systems, however, do use ad-hoc approaches for producing reasons.

The SUIF Explorer [6] and the ParaScope programming environment [1] are both interactive systems designed to help programmers to parallelize their programs. Both SUIF Explorer and ParaScope enable programmers to see the dependencies that inhibit the parallelization of a loop. While these dependencies can be helpful, they just tell the user that a dependency exists but not *why* it exists. The SUIF Explorer goes a step further by incorporating program slicing: the SUIF Explorer enables a user to look at the program slice of the references involved in dependencies. In a way, these program slices serve as "reasons" for dependencies: they suggest what code may be responsible for the dependences that inhibit parallelization. While slices are invaluable they can often be large; if a slice is large, the user may have no idea for where to look within the slice in order to resolve the dependencies. The SUIF Explorer can benefit from our approach: our approach would, for example, enable the user to ask why a particular statement is in the slice.

We can obtain some reasons for analysis failure by implementing our analyses in a declarative language, such as Prolog or Datalog. Prolog produces a trace that captures why something failed. However, these systems would produce one reason for failure and not the full reason; i.e., the reasons are not sufficient. In addition, many declarative languages have difficulties with cyclic dependencies between rules. They also do not restrict their language to positive logic: This prevents them from computing useful reasons.

The FALCON system [2] system enables programmers to manually select a region of code to which certain optimizations should be applied. FALCON then attempts to apply the optimizations. Unlike our system, FALCON does not provide any reasons if the optimizations are inapplicable.

Refactoring tools, such as Eclipse [10], allow programmers to transform their programs by picking refactorings from a menu. These tools implement reasons in an ad-hoc way: each refactoring does its own checking for legality and reports problems to the user as it encounters them. Since the reason checking is ad-hoc, there is no reason to believe that the reasons are actually good enough.

For example, most Eclipse refactorings are optimistic; i.e., their associated analyses do not fully check that a refactoring is legal. Thus, even if a user addresses a reason, there is no guarantee that the enabled refactoring will actually be correct.

The structure of our analysis language is related to the way program analyses are expressed with the help of model checking. Steffen and Schmidt [9] show that program analyses can be performed using model checking. Their work expresses optimizations using the modal mu-calculus, resulting in formulas that are similar to the analyses in our approach. Our approach also performs a kind of model checking: The reasons denote a generalization of those models that cause the optimization to not apply. Therefore, Steffen and Schmidt's work faces some of the same challenges as ours. For example, Schmidt notes [8] that many analyses are used as their duals, citing the example of live variable analysis. We also often find it convenient to rephrase meet analyses as join analyses in order to get effective reasons.

As in our system, the Cobalt system [5] also requires users to express their analyses in a special purpose language; Cobalt's goal is to prove the correctness of optimizations. There are some significant similarities in how one expresses the safety condition of optimizations in Cobalt and in our system. For example users express safety conditions in Cobalt as predicates that must hold in the code region between enabling statements and the statement affected by the optimization; our analyses are expressed similarly except that in our system the *forall* is explicit while in Cobalt it is implicit. Unlike our system, Cobalt does not require analyses to be expressed in positive logic; this restriction enables us to compute useful reasons for analysis failure.

To summarize, we are the first to come up with a general approach for computing reasons. While some prior systems attempt to produce reasons, they use an ad-hoc approach. As a consequence their reasons may not be necessary or sufficient. We believe that our approach can be beneficial to interactive transformation systems.

12. Conclusion

As programs and programming languages become increasingly complex, we will increasingly find that many desirable transformations are inapplicable and many analyses produce undesirable results. For example, the presence of dynamic class loading and reflection in Java degrades the results of both analyses and transformations [4]. Thus, we anticipate that many analysis and transformation tools will need input from the user. To obtain this input, the analysis and transformation tools must be able to produce reasons that guide the user in giving the needed input. In this paper we described and evaluated an approach for producing such reasons.

To produce such reasons, we have designed a language that captures parts of program analyses that degrade results; the analysis writer can write the remaining parts of the analysis using any method. We give two semantics for the language; the value semantics evaluates analyses in the language and the reason semantics produces reasons when analyses produce undesirable results.

We evaluate our approach by implementing a number of analyses in our language and running them on a number of standard Java benchmarks. We show that our approach produces reasons that are typically small (and thus probably useful to the user) and it takes, on average, 0.01 seconds to produce a reason (and thus fast enough for interactive use). We also demonstrate theoretical properties about our approach; specifically, we show that our reasons for an analysis are necessary and sufficient for ensuring the success of the analysis and that our analysis language can express arbitrary data-flow analyses.

13. Acknowledgements

We would like to thank Christoph Reichenbach, Mike Hind, Clayton Lewis, Darko Stefanovic and Mark Marron for their many helpful suggestions.

References

[1] Keith D. Cooper, Mary W. Hall, Robert T. Hood, Ken Kennedy, Kathryn S. McKinley, John M. Mellor-Crummey, Linda Torczon, and Scott K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.

[2] Luiz DeRose, Kyle Gallivan, Efstratios Gallopoulos, Bret A. Marsolf, and David A. Padua. FALCON: A MATLAB interactive restructuring compiler. In *Languages and Compilers for Parallel Computing*, pages 269–288, 1995.

[3] Jan Friso Groote and Misa Keinänen. Solving disjunctive/conjunctive boolean equation systems with alternating fixed points, 2004.

[4] Martin Hirzel, Daniel von Dincklage, Amer Diwan, and Michael Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2):11, 2007.

[5] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations, 2003.

[6] Shih-Wei Liao, Amer Diwan, Robert P. Bosch Jr., Anwar M. Ghuloum, and Monica S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Principles Practice of Parallel Programming*, pages 37–48, 1999.

[7] Angelika Mader. *Verification of Modal Properties Using Infinite Boolean Equation Systems*. PhD thesis.

[8] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98*, pages 38–48, New York, NY, USA, 1998. ACM Press.

[9] David A. Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. In *SAS*, pages 351–380, 1998.

[10] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developers Guide to Eclipse*. Addison-Wesley, May 2003.

[11] Standard performance evaluation corporation. SPECjbb2000 (java business benchmark). http://www.spec.org/jbb2000.

[12] Standard performance evaluation corporation. SPECjvm98 benchmarks. http://www.spec.org/jvm98.

[13] M. Weiser. Program slicing. In *Proceedings of ICSE*, pages 439–449. IEEE Computer Society Press, 1981.