Thesis for the Degree of Licentiate of Engineering

Static Analysis and Deductive Verification of Programs

Tobias Gedell

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Göteborg, 2006

Static Analysis and Deductive Verification of Programs
Tobias Gedell

© Tobias Gedell, 2006

Technical Report no. 15L
ISSN 1652–876X
Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Printed at Chalmers, Göteborg, 2006

Abstract

This thesis is concerned with analysis of programs. Analysis of programs can be divided into two camps: static analysis and formal verification.

Static program analyses compute a result and terminate for all programs. Since virtually all interesting semantic properties are undecidable, a static program analysis needs to be approximative to ensure termination. When designing such an analysis it can be hard to know which features that have the largest impact on the precision and should be added. This is the subject of the first paper in this thesis in which we investigate the impact a number of features have on the precision of usage analysis.

Formal verification often refers to deductive verification based on logic and theorem proving. When verifying a property, the program and the property are both translated into logical formulas and a theorem prover is used to show that the property holds for the program. Formal verification is a much more precise and general purpose technique than static analysis. This does, however, not come for free. It is extremely hard to find good heuristics for guiding the automatic construction of proofs. Therefore, user interaction is often required which makes the verification very time consuming and expensive.

Static program analysis is limited by its approximative nature and program verification by its high cost. It is, therefore, interesting to try to combine the strengths of the two techniques. This can be done in both directions: by letting a static program analysis use a theorem prover designed for program verification or letting a program verifier use a static program analysis. The latter combination is the subject of the second and third paper in this thesis.

We make the following contributions:

- We investigate the impact of a number of features on the precision of usage analysis.
- We show how a static program analysis can be embedded into a theorem prover.
- We show how interactive techniques for handling loops can sometimes be made automatic by using a dependence analysis.

Acknowledgments

First of all I would like to thank my supervisor Reiner Hähnle for all the help, support and encouragement he has given me. I would also like to thank my former co-supervisor Jörgen Gustavsson for all the time we have spent together and for being such a great person.

Secondly, I would like to thank my fellow PhD students for the nice working environment. I especially would like to thank Daniel Hedin, Ulf Norell, Philipp Rümmer, and Dennis Walter for all help and our frequent (non-)work related discussions.

Finally, I would like to thank all my other friends and my family.

Contents

Chapter 1

Introduction

This thesis is concerned with analysis of programs. Analysis of programs can be divided into two camps: static analysis and formal verification. In the following sections we introduce both these techniques and discuss why it is interesting to try to combine them.

1 Static Program Analysis

The aim of static program analysis is to statically determine various properties of programs. In this thesis, when talking about static program analysis we refer to the class of automatic and inexpensive program analyses that are often used to guide program optimizations or establish rather simple properties about programs. Two examples of static program analyses in this class are:

- **Type checking** - Determines whether a program is correct with respect to a given type system.
- **Null pointer analysis** - Analyzes each pointer variable in a program to determine whether it can, at runtime, contain a null pointer.

An important feature of these two program analyses, and program analyses in general, is that they compute a result and terminate for all programs. Unfortunately, virtually all interesting semantic properties are undecidable. That a property is undecidable can be proven by showing that computing the property would at the same time solve the halting problem. For the null pointer analysis, this can be shown by the following example where p is an arbitrary program and `x` a variable:

```
x = new Object();
p                                                    (1)
x = NULL;
```

If the null pointer analysis establishes that `x` will be assigned a null pointer, we know that p terminates and have, thus, solved the halting problem.[1]

It is clear that a static program analysis must be approximative in order to handle the termination requirement. For the example above, the null pointer analysis will say that `x` *might* contain a null pointer. When computing such an approximative result it is important that it is done in the right way.

Consider a static program analysis checking whether a security property is fulfilled by a particular program. Let us also assume that if the analysis finds that the program fulfills the security property, we will regard the program as being safe. In this case it is important that when the program analysis approximates, it does not mistake a program not fulfilling the security property for one that does. If it did, we could no longer trust the program analysis. On the other hand, it is acceptable that the program analysis mistakes a secure program for an insecure program. As a consequence, when the analysis rejects a program because it believes it is insecure, we do not know if it is because the program is inherently insecure or if it is due to the analysis approximating the result. Although undesired, this will not have the drastic consequences the opposite behavior would have.

When an analysis behaves in this desired way it is said to be *sound*. Designing a good approximating program analysis is a non-trivial task that requires a lot of thought. An approximation good for one kind of analysis might be terrible for another.

Another important feature of the class of static program analyses that we consider is that they are computationally cheap, i.e. that they are not too resource or time consuming. Often static program analyses have upper bounds of their complexity that can be formally proven. Ideally, a static program analysis should be close to linear at least in practice. This is very different from deductive program verification, that we will return to later, for which usually no termination guarantee exists.

When designing an analysis it can be hard to know which features have the largest impact on the precision and should be added to the analysis. There is a number of features that can be incorporated to make it more precise, but also at the same time more computationally expensive. Simply adding all features will probably render the analysis too expensive. This is the subject of the first paper in this thesis in which we investigate the impact a number of features have on the precision of a particular static analysis.

Because of the approximative nature of program analyses they are not very well suited to verify or establish complex properties. Most of the time they are tailor-made for computing rather simple properties in situations where the computational cost is of more importance than the precision of the computed results.

[1] Assuming that p itself does not refer to `x`.

1.1 Usage analysis

The particular static program analysis studied in the first paper in this thesis is *usage analysis* [LGH$^+$92, Mar93, TWM95, Gus98, WPJ99, WPJ00, GS00, Wan02]. Usage analysis works on programs written in lazy functional languages and is best explained by showing how lazy evaluation works.

The main feature of lazy evaluation is that expressions should not be evaluated before they are needed and that an expression should only be evaluated once, i.e. its value should be shared by all its successive uses. Consider the following example.

```
let   x = 1 + 2
      y = 3 + 4                                         (2)
in    x + x
```

When evaluating the expression, the computations for `x` and `y` will be stored unevaluated in the program memory, called the *heap*. We will refer to expressions stored in the heap as *closures*. When evaluating the expression `x + x`, we fetch the closure for `x` from the heap and evaluate it. When the expression has been evaluated we make sure that we update its closure with the result. This way we make sure that when `x` is used the second time, we fetch the already computed result from the heap and do not recompute the expression. It is important to notice that since `y` was never needed its value was never evaluated.

Lazy evaluation allows the programmer to focus on what should be computed instead of in which order the computation should be done. It also allows for the use of infinite structures such as infinite lists which cannot be used in languages with strict evaluation semantics.

One inefficiency of lazy evaluation is that the updating of evaluated closures is not always needed. If we change the expression in (2) to `x + y` instead of `x + x`, the unnecessary overhead is illustrated. Now both `x` and `y` will only be used once which means that the time spent updating their closures is wasted. How serious this is depends, of course, on how often expressions are only used once in functional programs. Measurements by Marlow [Mar93] have shown that for a particular Haskell implementation as many as 70% of all updates are unnecessary and that these updates stand for up to 20% of the total running time of a program.

If we knew which expressions are only used once during the execution of a program we could use this information to avoid updating their closures. This would make the programs run faster. Besides avoiding updates the information can also be used to enable a number of optimizing program transformations such as inlining, let floating, and full laziness [PJPS96]. The output of the usage analysis is an annotated version of the analyzed program. Each point in the annotated program that allocates closures is annotated with 1 or ω. If a point is annotated with 1, it means that all closures created at that point is only used once.

The usage analysis in the first paper in this thesis is defined as a type based program analysis. A type based program analysis can be seen as an extension to the underlying type system where the typing rules are extended to not only infer regular types but also usage information.

2 Program Verification

When talking about program verification in this thesis we mean deductive verification based on logic and theorem proving. In a program verifier of this type, the semantics of the target language is expressed in a program logic. Any property that can be expressed using the logic can then be checked for a particular program. When verifying a property, the program and the property are both translated into logical formulas and a theorem prover is used to show that the property holds for the program.

This is a much more precise and general purpose technique than the previously introduced static program analysis. It is more precise because we encode the exact semantics of the target language and use it to reason about programs in a precise way. It is general purpose since it can handle all properties that can be expressed in its logic. This does, however, not come for free. Having an expressive logic means that it is impossible to find a complete calculus for it. It will be theoretically impossible to establish correctness of all valid properties. It also becomes extremely hard to find good heuristics for guiding the automatic construction of proofs. Therefore, user interaction is often required which makes the verification very time consuming and expensive. It also forces the user to have a good knowledge of how the underlying program logic and calculus works.

Deductive program verification often does not have any termination guarantee. For the example in (1), a program verifier would in general be unable to prove or refute the property and, thus, never terminate.

Since a program verifier is made to be general it will not be optimized for any special class of properties. There will, therefore, be a rather large overhead when reasoning using the program logic instead of using a tailor-made static program analysis.

The main advantage of using program verification is that it can handle much more complex properties. Often, deductive program verification is used to verify that programs meet their functional specifications.

2.1 The KeY tool

The program verifier used in this thesis is the KeY tool [ABB$^+$05], which features an interactive theorem prover for formal verification of sequential Java programs. In KeY the program to be verified and the property are modeled in a dynamic logic called Java DL [Bec01]. The dynamic logic is a modal logic in which Java programs can occur as parts of formulas using modality operators.

The formula $\langle \texttt{p} \rangle\, \phi$ expresses that the program `p` terminates, without throwing an exception, in a state in which ϕ holds. A formula $\phi \;\rightarrow\; \langle \texttt{p} \rangle\, \psi$ is valid if for every state $\mathcal{S}$, satisfying precondition ϕ, a run of the program `p` starting in $\mathcal{S}$ terminates normally, and in the terminating state the postcondition ψ holds.

Deduction in the JAVA DL sequent calculus is based on symbolic program execution and simple program transformations. The rules of the calculus, called taclets, are implemented in a domain specific tactic language. A taclet typically consists of a guard pattern that is matched against the sequent under consideration and an action which is performed if the taclet is applied.

3 Combining Program Analysis and Verification

Static program analysis and deductive program verification are similar. Despite being two extremes on an abstract axis of precision, they are both working with exactly the same thing: properties of programs. There are, however, differences. One of these is that they target different classes of properties that have different requirements. Static program analysis is limited by its approximative nature and program verification by its high cost. It is, therefore, interesting to try to combine the strengths of the two techniques. This can be done in both directions: by letting a static program analysis use a theorem prover designed for program verification or by letting a program verifier use a static program analysis.

An example of the former could be a security analysis which needs to prove that for some loops certain invariants hold. This is something which is difficult for an automatic static program analysis to do. Normally, one would let the analysis approximate the results but it could render the analysis too imprecise to be useful. In cases like these it is possible that we can benefit from letting the analysis use a theorem prover to reason about the loop. We would loose the termination guarantee but it might in practice be worth it to get a more precise result.

In the latter case it also appears like there is much to be gained. Since we are dealing with a technique that requires interaction we do not loose anything by adding a cheap and automatic static program analysis. An example of this could be a program verifier having a set of general rules for handling almost any kind of properties but where each rule is rather costly to use. When verifying a program, the verifier will be forced to use the costly rules even for properties that are simple enough to be handled by a computationally cheap static program analysis. Consider the following method call: `o.m();`. Upon reaching the call to `o.m()` a program verifier probably needs to ensure either that `o` never contains a null pointer or handle the case when a null pointer exception is thrown. Using the rules of the verifier to prove that the variable never contains a null pointer can be very costly. Handling the case when an exception is thrown might also be very costly. Using a null pointer analysis to quickly decide whether the variable can contain a null pointer would enable us to sometimes avoid the exception case when it cannot be reached. It does not matter that the static analysis is imprecise, the default behavior of the program verifier corresponds to using a static analysis that always answers that the variable can contain a null pointer.

As long as the time spent running the null pointer analysis is less than the time saved by not having to handle unnecessary cases, we clearly benefit from doing this.

Combining static program analysis and program verification can be done in at least the following two ways:

- The static program analysis and the program verifier are kept separate. First the analysis is run on the program under consideration and the computed results are given along with the program to the verifier. Here, there is no interaction between the analysis and verifier which limits the approach. It is, however, the easiest way of combining them.
- The program analysis and the program verifier can be made to interact with each other. Instead of only running the analysis once before the verifier starts, the analysis can be run whenever the verifier needs it. The verifier can then pass the analysis information about the program which could help the analysis to compute a more precise result. In this way the static analysis and the verifier are interleaved and can be run in an incremental fashion.

Combining static program analysis and program verification is the subject of both the second and third paper in this thesis. In the second paper we show how a program analysis can be embedded into the KeY tool. In the third paper we show how interactive techniques for handling loops can sometimes be made automatic by using a dependence analysis.

4 Overview

This thesis consists of three papers, each discussing a topic related to static program analysis or the combination of static program analysis and program verification.

4.1 Paper I: Polymorphism, Subtyping, Whole Program Analysis and Accurate Data Types in Usage Analysis

In this paper we study the impact of various features on a full scale implementation of a usage analysis for Haskell. The questions we investigate are: Should the analysis be monomorphic or have some degree of polymorphism? What about subtyping? How should the analysis deal with user defined algebraic data types? Should it be a whole program analysis?

When designing a usage analysis, a choice has to be made for each of these questions. Several researchers have speculated that these features are important but there has been a lack of empirical evidence. Since some of the features can be rather costly, it is important for designers of program analyses to know how much higher precision it is reasonable to expect by adding them.

In order to evaluate the above features, we have implemented a range of usage analyses with

- different degrees of polymorphism,
- with and without subtyping,
- different treatments of data types, and
- as whole program analyses and as modular analyses.

Our measurements show that all features increase the precision. It is, however, not necessary to have them all to obtain an acceptable precision.

4.2 Paper II: Embedding Static Analysis into Tableaux and Sequent based Frameworks

In this paper we present a method for embedding static analysis into tableaux and sequent based frameworks. In these frameworks, the information flows from the root node to the leaf nodes. We show that the existence of free variables in such frameworks introduces a bi-directional flow, which can be used to collect and synthesize arbitrary information.

We use the free variables mechanism and the tactic language in the KeY tool to implement a reaching definitions analysis. The chosen analysis is a common and well-known analysis that shows the potential of the method.

The achieved results are promising and open up for new areas of application of tableaux and sequent based theorem provers.

4.3 Paper III: Automating Verification of Loops by Parallelization

In this paper we show how one can replace interactive proof techniques such as induction, with automated first-order reasoning in order to deal with parallelizable loops. A loop can be parallelized whenever it avoids dependence of the loop iterations from each other.

Loops are a major bottleneck in formal software verification, because they generally require user interaction: typically, induction hypotheses or invariants must be found or modified by hand. This involves expert knowledge of the underlying calculus and proof engine.

We develop a dependence analysis that ensures parallelizability. It guarantees soundness of a proof rule that transforms a loop into a universally quantified update of the state change information represented by the loop body. This makes it possible to use automatic first order reasoning techniques to deal with loops. The method has been implemented in the KeY tool. We evaluated it with representative case studies from the Java Card domain.

5 Contributions

The main contributions of the work presented in this thesis are:

- Paper I
  - We perform a systematic evaluation of the impact of various features on usage analysis, giving valuable input to designers of usage analyses.
  - We have implemented and used a measuring technique that gives more relevant results than previous case studies.
- Paper II
  - We show how synthesis can be performed in a tableau or sequent style prover, which opens up for new areas of application.
  - We show how the rules of a program analysis can be embedded into a program logic and coexist with the original rules by using a tactic language.
  - We give a proof-of-concept of our method. We do this by giving the full embedding of a program analysis in the KeY tool.
- Paper III
  - We show how one can replace interactive proof techniques such as induction, with automated first-order reasoning in order to deal with parallelizable loops.
  - We show the feasibility of our technique by implementing it in the KeY tool.
  - We show the relevance of our technique by evaluating it with representative case studies from the Java Card domain.

Bibliography

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[GS00] Jörgen Gustavsson and Josef Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In Markus Mohnen and Pieter W. M. Koopman, editors, *IFL*, volume 2011 of *Lecture Notes in Computer Science*, pages 140–157. Springer, 2000.

[Gus98] Jörgen Gustavsson. A type based sharing analysis for update avoidance and optimisation. In *ICFP*, pages 39–50. ACM, SIGPLAN Notices 34(1), 1998.

[LGH+92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. L. Peyton Jones, and P. Wadler. Avoiding Unnecessary Updates. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, Glasgow, 1992. Springer.

[Mar93] S. Marlow. Update Avoidance Analysis by Abstract Interpretation. In *Proc. 1993 Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer–Verlag, 1993.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. of ICFP'96*, pages 1–12. ACM, SIGPLAN Notices 31(6), May 1996.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. of FPCA*, La Jolla, 1995. ACM Press, ISBN 0-89791-7.

[Wan02] Keith Wansbrough. *Simple Polymorphic Usage Analysis.* PhD thesis, Computer Laboratory, Cambridge University, England, March 2002.

[WPJ99] Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. In *Proc. of POPL'99.* ACM Press, 1999.

[WPJ00] Keith Wansbrough and Simon Peyton Jones. Simple Usage Polymorphism. In *ACM SIGPLAN Workshop on Types in Compilation.* Springer-Verlag, 2000.

Polymorphism, Subtyping, Whole Program Analysis and Accurate Data Types in Usage Analysis

Tobias Gedell   Jörgen Gustavsson   Josef Svenningsson

Abstract

There are a number of choices to be made in the design of a type based usage analysis. Some of these are: Should the analysis be monomorphic or have some degree of polymorphism? What about subtyping? How should the analysis deal with user defined algebraic data types? Should it be a whole program analysis?

Several researchers have speculated that these features are important but there has been a lack of empirical evidence. In this paper we present a systematic evaluation of each of these features in the context of a full scale implementation of a usage analysis for Haskell.

Our measurements show that all features increase the precision. It is, however, not necessary to have them all to obtain an acceptable precision.

1 Introduction

In this article we study the impact of polymorphism, subtyping, whole program analysis and accurate data types on type based *usage analysis*. Usage analysis is an analysis for lazy functional languages that aims to predict whether an argument of a function is used at most once. The information can be used to reduce some of the costly overhead associated with call-by-need and perform various optimizing program transformations.

Polymorphism Polymorphism is the primary mechanism for making a type based analysis context sensitive.

Previous work by Peyton Jones and Wansbrough has indicated that polymorphism is important for usage analyses. Convinced that polymorphism could be dispensed with they made a full scale implementation of a completely monomorphic usage analysis. However, it turned out that it was "almost useless in practice" [WPJ99]. They drew the conclusion that the reason was the lack of polymorphism. In the end they implemented an improved analysis with a simple form of polymorphism that also incorporated other improvements [Wan02]. The resulting analysis gave a reasonable precision but there is no evidence that polymorphism was the crucial feature.

In contrast to these indications, several studies on points-to analysis for C have shown that monomorphic analyses [FFA00b, HT01, FRD00, Das00, DLFR01] give adequate precision for the purpose of an optimizing compiler [DLFR01]. Moreover, extensions of these analyses seem to have only a moderate effect. For example, Foster et al [FFA00b] showed that adding polymorphism to Andersen's [And94] inclusion based points-to analysis for C only gave a moderate increase in precision and Das et al [DLFR01] came to the same conclusion when they added a limited degree of polymorphism to the analysis in [Das00].

A possible explanation for the indicated discrepancy is that functional programmers more often write small reusable functions because of the excellent features for abstraction. One of the goals of this work has been to confirm or refute this discrepancy.

Subtyping Another important feature in type based analyses is subtyping. It provides a mechanism for approximating a type by a less informative super type. This gives a form of context sensitivity since a type may have different super types at different call sites. It also provides a mechanism for combining two types, such as the types of the branches of an if expression, by a common super type. Thus, subtyping and polymorphism interfere with each other.

This raises a number of questions. Does it suffice with either polymorphism or subtyping? How much is gained by having the combination?

Whole program analysis Another issue that also concerns context sensitivity is whole program analysis versus modular program analysis. A modular analysis which considers each module in isolation must make a worst case assumption about the context in which it appears.

This will clearly degrade the precision of the analysis. But how much? Is whole program analysis a crucial feature? And how does it interact with the choice of monomorphism versus polymorphism?

Data types Another important design choice in a type based analysis is how to deal with user defined data types. The intuitive and accurate approach may require that the number of annotations on a type is exponential in the size of the type definitions of the analyzed program. The common solution to the problem is to limit the number of annotations on a type in some way, which leads to spurious loss of precision. The question is how big the loss is in practice.

Contributions In order to evaluate the above features, we have implemented a range of usage analyses with

- different degrees of polymorphism,
- with and without subtyping,
- different treatments of data types, and
- as whole program analyses and as modular analyses.

All analyses have been implemented in the GHC compiler and have been measured with GHC's optimizing program transformations both enabled and disabled.

Our systematic evaluation shows that each of these features has a significant impact on the precision of the analysis. Especially, it is clear that some kind of context sensitivity is needed through polymorphism or subtyping. Our results also show that the different features are intertwined and interfere with each other. The combined effect of polymorphism and subtyping is for example not very dramatic although each one of them has a large effect on the accuracy. Another example is that whole program analysis is more important for monomorphic analysis than polymorphic analysis.

Outline The paper is organized by considering each dimension in turn. We evaluate different degrees of polymorphism in Section 3, subtyping in Section 4, data types in Section 5 and whole program analysis in Section 6.

2 Usage Analysis

Implementations of lazy functional languages maintain sharing of evaluation by updating. For example, the evaluation of

$$(\lambda x.x + x)\,(1 + 2)$$

proceeds as follows. First, a closure for $1 + 2$ is built in the heap and a reference to the closure is passed to the abstraction. Second, to evaluate $x+x$ the value of x is required. Thus, the closure is fetched from the heap and evaluated. Third, the closure is updated (i.e., overwritten) with the result so that when the value of x is required again, the expression needs not be recomputed.

The same mechanism is used to implement lazy data structures such as potentially infinite lists.

The sharing of evaluation is crucial for the efficiency of lazy languages. However, it also carries a substantial overhead which is often not needed. For example, if we evaluate

$$(\lambda x.x + 1)\,(1 + 2)$$

then the update of the closure is unnecessary because the argument is only used once.

The aim of usage analysis is to detect such cases. The output of the analysis is an annotated program. Each point in the program that allocates a closure in the heap is annotated with 1 if the closure that is created at that point is always used at most once. It is annotated with ω if the closure is possibly used more than once or if the analysis cannot ensure that the closure is used at most once.

The annotations allow a compiler to generate code where the closures are not updated and thus effectively turning call-by-need into call-by-name. Usage analysis also enables a number of program transformations [PJPS96, JM99].

Usage analysis has been studied by a number of researchers [LGH$^+$92, Mar93, TWM95, Fax95, Gus98, WPJ99, WPJ00, GS00, Wan02].

2.1 Measuring the Effectiveness

We measured the effectiveness of the analyses by running them on the programs from the *nofib* suit [Par93] which is a benchmarking suit designed to evaluate the Glasgow Haskell Compiler (GHC). We excluded the toy programs and ran our analysis on the programs classified in the category *real* but had to exclude the following three programs: *HMMS* did not compile with GHC on our test system, *ebnf2ps* is dependent on a version of Happy that we could not get to work with our version of GHC, and *veritas* because many analyses ran out of memory when analyzing it.

Despite the name of the category, the average size of the programs is unfortunately quite small.

The notion of effectiveness When measuring the effectiveness it is natural to do so by modifying the runtime system of GHC. The runtime system is modified to collect the data needed to compute the effectiveness during a program's execution.

The easiest way is to count how many created closures that are only used once and how many of those closures that were detected by the analysis. This can be implemented by adding three counters to the runtime system: one that gets incremented as soon as an updatable closure is created, one that gets incremented each time a closure is used a second time, and one that gets incremented as soon as a closure annotated with 1 is created. With these counters one can compute an effectiveness of an analysis:

$$\frac{\textit{closures annotated with } 1}{\textit{created closures} - \textit{closures used twice}}$$

This is the measure used by Wansbrough [Wan02].

A drawback of this approach is that it does not take into account that each program point can only have one annotation – if any of the closures allocated at a program point is used more than once, that program point has to be annotated with ω for the analysis to be sound. Thus, if there is such a program point (and there typically are) then even a perfect analysis would not get a 100 percent effectiveness.

What we would like to do is to compute the effectiveness by measuring the proportion of *program points* that are correctly annotated instead of the proportion of *updates* that are avoided. We, therefore, modified the run time system to compute the best possible annotations which are consistent with the observed run time behavior. I.e., if all the closures allocated at a specific program point is used at most once during the execution, that program point could be annotated with 1 otherwise ω. We did this by, for each closure, keeping track of at which program point it was created. When a closure is used a second time we add its program point to the set of program points that need to be annotated with ω. We were careful to exclude code that was not executed in the executions such as parts of imported libraries which were not used. It is important to note that this way of measuring is still based on running the program on a particular input and a perfect analysis may still get an effectiveness which is less than 100 percent.

These two different ways of measuring differ also at another crucial point. The former measurement depends very much on how many times each program point that allocates closures is executed. If a single program point allocates a majority of all closures, the computed effectiveness will depend very much on whether that single program point was correctly annotated by the analysis. In contrast, the effectiveness computed with the latter measurement will hardly be affected by one conservative annotation.

We think that the latter notion of effectiveness is more informative and have, therefore, used it for all our measurements.

Optimizing program transformations Our implementation is based on GHC which is a state of the art Haskell implementation. GHC parses the programs and translates them into the intermediate language Core, which is essentially System F [PJPS96]. When GHC is run with optimizations turned on, it performs aggressive program transformation on Core before it is translated further. We inserted our analyses after GHC's program transformations just before the translation to lower level representations.

We ran the analysis with GHC's program transforming optimizations both enabled and disabled. The latter gives us a measure of the effectiveness of an analysis on code prior to program transformations. This is relevant because usage information can be used to guide the program transformations themselves.

3 Polymorphism

We start by evaluating usage polymorphism. Too see why it can be a useful feature, consider the function that adds up three integers.[1]

$$\mathit{plus3}\ x\ y\ z = x + y + z$$

Which usage type should we give to this function? Since the function uses all its arguments just once, it seems reasonable to give it the following type.

$$\mathit{Int}^1 \rightarrow \mathit{Int}^1 \rightarrow \mathit{Int}^1 \rightarrow \mathit{Int}^\omega$$

The annotations on the type express that all three arguments are used just once by the function and that the result may be used several times. However, this type is not correct. The problem is that the function may be partially applied:

$$\mathit{map}\ (\mathit{plus3}\ (1+2)\ (3+4))\ \mathit{xs}$$

[1]This example is due to Wansbrough and Peyton Jones [WPJ00]

If xs has at least two elements then $\mathit{plus3}\,(1+2)\,(3+4)$ is used more than once. As a consequence, so is also $(1+2)$ and $(3+4)$.

To express that functions may be used several times we need to annotate also function arrows. A possible type for *plus3* could be:

$$\mathit{Int}^{\omega} \to^{\omega} \mathit{Int}^{\omega} \to^{\omega} \mathit{Int}^{1} \to^{\omega} \mathit{Int}^{\omega}$$

The function arrows are annotated with ω which indicates that *plus3* and its partial applications may be used several times. The price we pay is that the first and the second argument are given the type Int^{ω}. This type is sound but it is clearly not a good one for call sites where *plus3* is not partially applied. What is needed is a mechanism for separating call sites with different usage.

The solution to the problem is to give the function a usage polymorphic type:

$$\forall\, u_0\, u_1\, u_2\, u_3 \mid u_2 \leq u_0, u_3 \leq u_0, u_3 \leq u_1 . \mathit{Int}^{u_0} \to^{\omega} \mathit{Int}^{u_1} \to^{u_2} \mathit{Int}^{1} \to^{u_3} \mathit{Int}^{\omega}$$

The type is annotated with usage variables and the type schema contains a set of constraints which restrict how the annotations can be instantiated. A constraint $u \leq u'$ simply specifies that the values instantiated for u must be smaller than or equal to the values instantiated for u' where we have the ordering that $1 < \omega$. This form of polymorphism is usually referred to as constrained polymorphism or bounded polymorphism.

In our example, $u_2 \leq u_0$ enforces that if a partial application of *plus3* to one argument is used more than once then that first argument is also used more than once. Similarly, $u_3 \leq u_0$ and $u_3 \leq u_1$ makes sure that if we partially apply *plus3* to two arguments and use it more than once then both these arguments are used more than once.

3.1 Degrees of Polymorphism

There are many different forms of parametric polymorphism. In this paper we consider three different systems where usage generalization takes place at let-bindings.

- An analysis with monomorphic recursion in the style of ML. Intuitively, this gives the effect of a monomorphic analysis where all non-recursive calls have been unwound.
- An analysis with polymorphic recursion [Myc84, Hen93, DHM95]. Intuitively, this gives the effect of the previous analysis where recursion has been (infinitely) unwound.
- An analysis where the form of type schemas are restricted so that generalized usage variables may not be constrained. A consequence of the restriction is that an implementation need not instantiate (i.e., copy) a potentially large constraint set whenever the type is instantiated. Wansbrough and Peyton Jones [WPJ00] suggested this in the context of usage analysis and called it *simple usage polymorphism*.

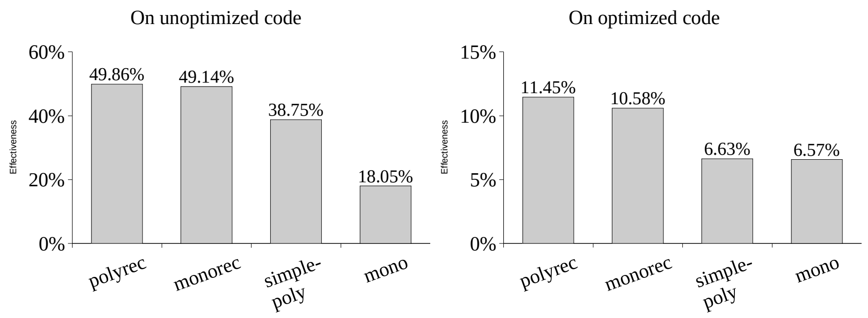

Figure 1: Measurements of polymorphism

With simple usage polymorphism it is not possible to give *plus3* the type

$$\forall u_0 u_1 u_2 u_3 | u_2 \leq u_0, u_3 \leq u_0, u_3 \leq u_1 . \mathit{Int}^{u_0} \rightarrow^{\omega} \mathit{Int}^{u_1} \rightarrow^{u_2} \mathit{Int}^{1} \rightarrow^{u_3} \mathit{Int}^{\omega}$$

because the generalized variables u_0, u_1, u_2, u_3 are all constrained. Instead we can give it the type

$$\forall\, u . \mathit{Int}^{u} \rightarrow^{\omega} \mathit{Int}^{u} \rightarrow^{u} \mathit{Int}^{1} \rightarrow^{u} \mathit{Int}^{\omega}$$

where we have unified the generalized variables into one. This type is clearly worse but it gives a degree of context sensitivity. An alternative is to give it a monomorphic type. For example

$$\mathit{Int}^{\omega} \rightarrow^{\omega} \mathit{Int}^{1} \rightarrow^{\omega} \mathit{Int}^{1} \rightarrow^{1} \mathit{Int}^{\omega}.$$

These types are incomparable and an implementation needs to make a heuristic choice. We use the heuristic proposed by Wansbrough [Wan02] to generalize the types of all exported functions and give local functions monomorphic types.

The analyses include usage subtyping; use an aggressive treatment of algebraic data types and are compatible with separate compilation (i.e., we analyze the modules of the program one by one in the same order as GHC). We discuss and evaluate all these features later on.

3.2 Evaluation

The results are shown in Figure 1, which shows the average effectiveness of each analysis, and Section A.1, which shows the effectiveness for each program.

The most striking observation is that the results are very different depending on whether GHC's optimizing program transformations are turned on or off. The effectiveness is much lower with program transformations turned on. We believe that an explanation of this is that GHC inlines many function calls. There is no need to create closures for the arguments of these function calls anymore and thus many targets for the analysis disappears. The net effect is that the proportion of difficult cases (such as closures in data structures and calls to unknown functions) increases which reduces the effectiveness.

Another explanation is strictness analysis [Myc82]. Strictness analysis can decide that the argument of a function is guaranteed to be used at least once (in any terminating computation). In those cases there is no need to suspend the evaluation of that argument. If an argument is used exactly once then it is a target for both strictness and usage analysis. When the strictness analysis (as part of GHC's program transformation) is ran first it removes some easy targets.

Another phenomena is that the benefits of polymorphism are smaller when program transformations are turned on. This is what you would expect since inlining naturally makes context sensitivity less important.

The results also show that the polymorphic analyses are significantly better than the monomorphic one. Polymorphic recursion turns out to have hardly any effect compared to monomorphic recursion. Simple polymorphism comes half way on unoptimized code – it is significantly better than monomorphism but significantly worse than constrained polymorphism, which shows that it can serve as a good compromise. This is, however, not the case for optimized code.

The largest surprise to us was that the accuracy of the monomorphic analysis is relatively good. This seems to contradict the results reported by Wansbrough and Peyton Jones [WPJ00] who implemented and evaluated the monomorphic analysis from [WPJ99]. They found that the analysis was almost useless in practice and concluded that it was the lack of polymorphism that caused the poor results. We do not have a satisfactory explanation for this discrepancy.

4 Subtyping

Consider the following code fragment.

$$\mathbf{let}\, x =^{u} 1 + 2\, \mathbf{in}\ \ldots$$

Here u is the usage annotation associated with the closure for $1 + 2$.

The analysis can take u to be 1 if and only if x is used at most once. That is assured by giving x the type Int^1. The type system then makes sure that the program is well typed only if x is actually used at most once.

If we on the other hand take u to be ω then x has the type Int^ω. It is always sound to annotate a closure with ω regardless of how many times it is used. We, therefore, want the term to be well typed regardless of how many times x is actually used. The solution is to let Int^ω be a *subtype* of Int^1. That is, if a term has the type Int^ω we may also consider it to have the type Int^1.

Subtyping makes the system more precise. Consider the function f.

$$f\ x\ y\ = \mathbf{if}\, x * x > 100\, \mathbf{then}\, x\, \mathbf{else}\, y$$

It seems reasonable that we should be able to give it, for example, the type

$$Int^\omega \to^\omega Int^1 \to^\omega Int^1.$$

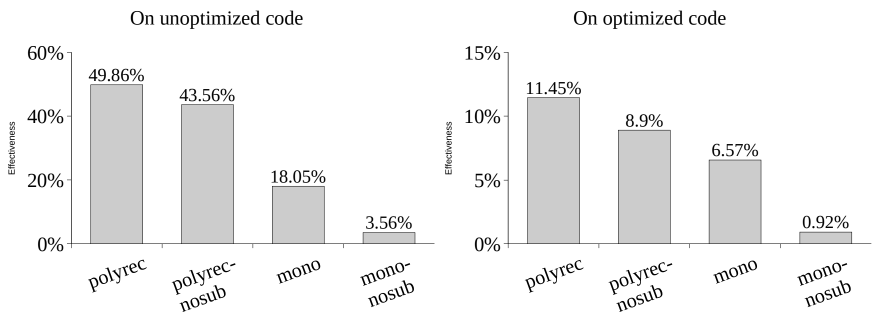

Figure 2: Measurements of subtyping

This type expresses that if the result of the function is used at most once then the second argument is used only once. The first argument is, however, used at least twice regardless of how many times the result is used.

To derive this type we must have usage subtyping. Otherwise, the types of the branches of the conditional would be incompatible – x has type Int^{ω} and y has the type Int^{1}. With subtyping we can consider x to have the type Int^{1}.

Without subtyping x and y has to have the same type and the type of the function must be

$$Int^{\omega} \rightarrow^{\omega} Int^{\omega} \rightarrow^{\omega} Int^{\omega}$$

which puts unnecessary demands on y.

Subtyping can also give a degree of context sensitivity. Consider, for example, the following program.

$$\begin{array}{ll} \textbf{let} & f\ x = x + 1 \\ & a = 1 + 2 \\ & b = 3 + 4 \\ \textbf{in} & f\ a + f\ b + b \end{array}$$

Here, b is used several times and is given the type Int^{ω}. Without subtyping nor polymorphism we would have to give a the same type and the two call sites would pollute each other.

When subtyping is combined with polymorphism it naturally leads to constrained polymorphism. Note, however, that subtyping is not the only source of inequality constraints in a usage analysis. Inequality constraints are also used for the correct treatment of partial application (see Section 3) and data structures. Thus, we use constrained polymorphism also in the systems without subtyping.

4.1 Evaluation

We have evaluated two systems without subtyping – a polymorphicly recursive and a monomorphic analysis. Both analyses use an aggressive treatment of data types and are compatible with separate compilation. Figure 2 shows the average effectiveness of each analysis. Section A.2 shows the effectiveness for each program. We have included the system with polymorphic recursion and subtyping and the monomorphic system with subtyping from Section 3 for an easy comparison.

The results show that the accuracy of the monomorphic system without subtyping is poor. The precision is dramatically improved if we add subtyping or polymorphism. Our explanation is that both polymorphism and subtyping gives a degree of context sensitivity which is crucial.

The polymorphic system without subtyping is in principle incomparable to the monomorphic system with subtyping. However, in practice the polymorphic system outcompetes the monomorphic one. The difference is much smaller when the analyses are run on optimized code which is consistent with our earlier observation that context sensitivity becomes less important because of inlining.

The combination of subtyping and polymorphism has a moderate but significant effect when compared to polymorphic analysis without subtyping. The effect is relatively larger on optimized code. The explanation we can provide is that the proportion of hard cases - which requires the combination – is larger because the optimizer has already dealt with many simple cases.

5 Algebraic data types

An important issue is how to deal with data structures such as lists and user defined data types. In this section we evaluate some different approaches.

Let us first consider the obvious method. The process starts with the user defined data types which only depend on predefined types. Suppose T is such a type.

$$data\, T\, \vec{\alpha} = C_1\, \vec{\tau} \mid \ldots \mid C_n\, \vec{\tau}$$

The types on the right hand side are annotated with fresh usage variables. If there are any recursive occurrences they are ignored. The type is then parameterized on these usage variables, $\vec{u}$.

$$data\, T\, \vec{u}\, \vec{\alpha} = C_1\, \vec{\tau}'_1 \mid \ldots \mid C_n\, \vec{\tau}'_n$$

Finally, any recursive occurrence of T is replaced with $T\, \vec{u}$. The process continues with the remaining types in the type dependency order and when T is encountered it is replaced with $T\, \vec{u}'$ where $\vec{u}'$ is a vector of fresh variables. If there are any mutually recursive data types they are annotated simultaneously.

As an example consider the following data type for binary trees:

$$data\ Tree\ \alpha = Node\ (Tree\ \alpha)\ (Tree\ \alpha) \mid Leaf\ \alpha$$

When annotated, it contains three annotation variables:

$$\begin{array}{lcl} data\ Tree\ \langle k_0, k_1, k_2\rangle\ \alpha & = & Node\ (Tree\ \langle k_0, k_1, k_2\rangle\ \alpha)^{k_0}\ (Tree\ \langle k_0, k_1, k_2\rangle\ \alpha)^{k_1} \\ & \mid & Leaf\ \alpha^{k_2} \end{array}$$

This approach is simple and accurate and we used it in all the analyses in the previous sections. The net effect is equivalent to a method where all non-recursive occurrences in a type are first unwound. As a result the number of annotation variables can grow exponentially. An example of this is the following data type:

$$\begin{aligned}
&\mathit{data}\, T_0\, \langle k_0 \rangle \ = C\ \mathit{Int}^{k_0} \\
&\mathit{data}\, T_1\, \langle k_0, k_1, k_2, k_3 \rangle = C'\ (T_0\, \langle k_1 \rangle)^{k_0} \mid C''\ (T_0\, \langle k_3 \rangle)^{k_2} \\
&\dots \\
&\mathit{data}\, T_n\, \langle k_0, \dots, k_m \rangle = C'_n\ (T_{n-1}\, \langle \dots \rangle)^{k_0} \mid C''_n\ (T_{n-1}\, \langle \dots \rangle)^{k_{m/2}}
\end{aligned}$$

Here T_n will contain 2^n usage variables.

In practice, the number of required variables sometimes grows very large. The largest number we have encountered was a type in the Glasgow Haskell Compiler which required over two million usage annotations. As a consequence a single subtyping step leads to over two million inequality constraints and our implementation simply could not deal with all those constraints. This problem was the reason for why we had to exclude the program veritas from our study. It is clear that an alternative is needed and we tried two different ones.

The first approach was to put a limit on the number of usage variables which are used to annotate a type. If the limit is exceeded then we simply use each variable several times on the right hand side of the type. We do not try to do anything clever and when we exceed the limit we simply recycle the variables in a round robin manner. This approach leads to ad-hoc spurious behavior of the analysis when the limit is exceeded but maintains good accuracy for small types. We tried this approach with a limit of 100, 10 and 1.

The second approach was to simply annotate all types on the right hand side with only ω. The effect is that information is lost when something is inserted into a data structure – the analysis simply assumes the worst about its usage. Intuitively this can be thought of as a special case of the approach above where the limit is zero.

All the analyses used for measuring the treatment of data types have subtyping and polymorphic recursion and are compatible with separate compilation.

5.1 Evaluation

The average effectiveness of each analysis is shown in Figure 3. In Section A.3 the effectiveness for each program is shown.

The results are quite different for optimized and unoptimized code. In the case of unoptimized code there is a clear loss in precision when we limit the number of annotation variables. The loss is quite small when the limit is 100 but quite dramatic when the limit is only 10. Going further and annotating with only one or no variables has a smaller effect.

The situation is different for optimized code. Here there is only a small difference when the number of variables are limited to 100 or 10. But there is a noticeable effect when one or no variables are used.

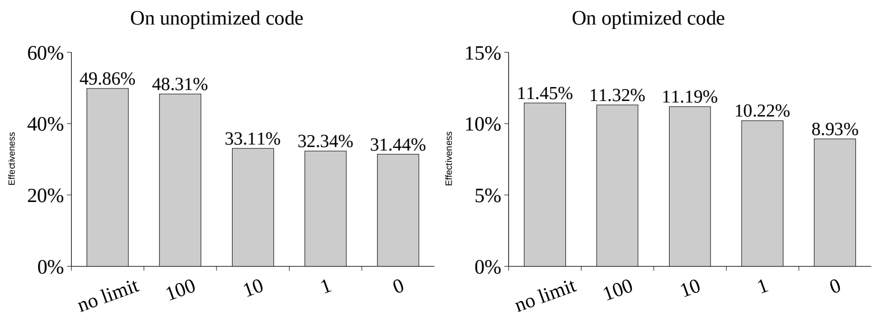

Figure 3: Measurements of treatments of data types

We believe that this effect stems from Haskell's class system. When Haskell programs are translated into Core each class context is translated to a so called dictionary parameter. A dictionary is simply a record of the functions in an instance of a class. Large classes leads to large records of functions which are passed around at run time. When the number of annotations are limited, it substantially degrades the precision for these records. Presumably, most dictionaries require more than 10 variables but less than 100 which explains the effect for unoptimized code.

These records are often eliminated by GHC's program transformations which specializes functions for each particular instance in a form of partial evaluation [Jon94, Aug93]. Thus, in optimized code there are not so many large types which explains why the effect of limiting the number of variables to 10 is quite small. When the limit on the other hand is one or zero it strikes all user defined types which has a significant effect.

6 Whole Program Analysis

So far all the analyses have been compatible with separate compilation. In this section we consider whole program analysis.

Suppose that f is an exported library function where the closure created for x' is annotated with u.

$$f\ x = \textbf{let}\ x' =^u x + 1\ \textbf{in}\ \lambda y.x' + y$$

In the setting of separate compilation we have to decide which value u should take without knowledge of how f is called. In the worst case, f is applied to one argument and the resulting function is applied repeatedly. The closure of x' is then used repeatedly so we must assume the worst and let u be equal to ω. We can then give f the type

$$Int^1 \rightarrow^\omega Int^1 \rightarrow^\omega Int^\omega$$

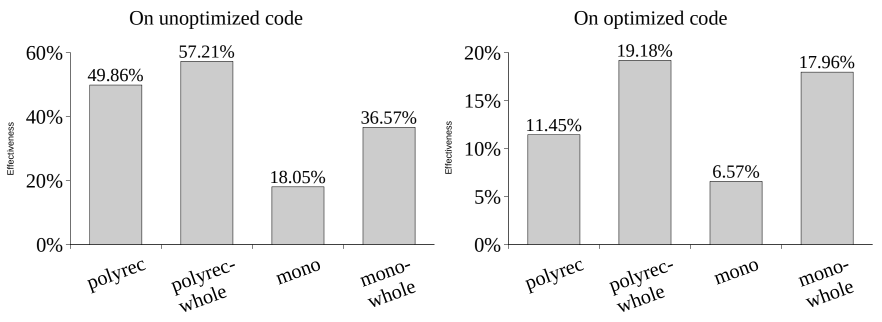

Figure 4: Measurements of whole program analysis

With separate compilation we must make sure that the types of exported functions are general enough to be applicable in all contexts. That is, it must still be possible to annotate the remaining modules such that the resulting program is well typed. Luckily, this is always possible if we ensure that the types of all exported functions have an instance where the positive (covariant) positions in the type are annotated with ω. In the type of f this is reflected in that the function arrows and the resulting integer are annotated with ω. Wansbrough and Peyton Jones [WPJ00] calls this process pessimization. Further discussion can be found in Wansbrough's thesis [Wan02].

In the setting of whole program analysis this process in unnecessary which improves the result of the analysis. We have chosen to evaluate the effect on two analyses, the polymorphicly recursive analysis with subtyping and the monomorphic analysis with subtyping. Both analyses use the aggressive treatment of data types.

6.1 Evaluation

The average effectiveness for each analysis is shown in Figure 4. Section A.4 shows the effectiveness for each program. They show that whole program analysis improves both analyses significantly on both unoptimized and optimized code.

The effect is greater for the monomorphic analysis. The explanation is that the inaccuracies that are introduced by the pessimization, needed for separate compilation, spreads further in the monomorphic analysis due to the lack of context sensitivity. One can think of pessimization as simulating the worst possible calling context which then spreads to all call sites.

An interesting observation is that there is only a small difference between the polymorphic and the monomorphic whole program analysis for optimized code. The combination of aggressive inlining and whole program analysis almost cancels out the effect of polymorphism.

7 Related Work

The usage analyses in this paper build on the type based analyses in [TWM95, Gus98, WPJ99, WPJ00, GS00, Wan02]. The use of polymorphism in usage analysis was first sketched in [TWM95] and was developed further in [GS00] and [WPJ00, Wan02] where simple polymorphism was proposed. Usage subtyping was introduced in [Gus98, WPJ99]. The method for dealing with data types was suggested independently by Wansbrough [Wan02] and ourselves [Ged03]. The method for dealing with separate compilation is due to Wansbrough and Peyton Jones [WPJ99].

The measurements of Wansbrough and Peyton Jones on their monomorphic analysis with subtyping and a limited treatment of data types showed that is was ”almost useless in practice”. Wansbrough later made thorough measurements of the precision of simple usage polymorphism with some different treatments of data types in [Wan02]. He concludes that the accuracy of the simple usage polymorphism with a good treatment of data types is reasonable which is consistent with our findings. He also compares the accuracy with a monomorphic usage analysis but the comparison is incomplete – the monomorphic analysis only has a very coarse treatment of data types.

Foster et al [FFA00a] evaluate the effect of polymorphism and monomorphism on Steensgaard's equality based points-to analysis [Ste96] as well as Andersen's inclusion based points-to analysis [And94]. Their results show that the inclusion based analysis is substantially better than the unification based. Adding polymorphism to the equality based analysis also has a substantial effect but adding polymorphism to the inclusion based analysis gives only a small improvement.

There are clear analogies between Steensgaard's equality based analysis and usage analysis without subtyping. Andersen's inclusion based analysis relates to usage analysis with subtyping. Given these relationships, our results are consistent with the results of Foster et al with one exception – the combination of polymorphism and subtyping has a significant effect in our setting. However, when we apply aggressive program transformations prior to the analysis and run it in whole program analysis mode then our results coincide.

8 Conclusions

We have performed a systematic evaluation of the impact on the accuracy of four dimensions in the design space of a type based usage analyses for Haskell. We evaluated

- different degrees of polymorphism: polymorphic recursion, monomorphic recursion, simple polymorphism and monomorphism,
- subtyping versus no subtyping,
- different treatments of user defined types, and

- whole program analysis versus analysis compatible with separate compilation.

Our results show that all of these features individually have a significant effect on the accuracy. A striking outcome was that the results depended very much on whether the analyzed programs were first subject to aggressively optimizing program transformations.

Our evaluation of polymorphism and subtyping showed that the polymorphic analyses clearly outperform their monomorphic counterparts. The effect was larger when the analyses did not incorporate subtyping. This is not surprising given that subtyping gives a degree of context sensitivity and, thus, partially overlaps with polymorphism. Polymorphic recursion turned out to give very little when compared to monomorphic recursion. For unoptimized code, simple polymorphism (where variables in types schemas cannot be constrained) was shown to lie in between monomorphism and constrained polymorphism.

The measurements also showed that the treatment of data types is important. The effectiveness of the different alternatives turned out to depend on whether the code was optimized or not. We believe that the explanation is coupled to the implementation of Haskell's class system and, thus, that this observation might be rather Haskell specific.

Whole program analysis turned out to have a rather large impact. The effect was greater for monomorphic analysis. The reason is that the conservative assumptions, that have to be made in the setting of separate compilation, have larger impact due to the lack of context sensitivity in monomorphic analysis. In fact, the whole program monomorphic analysis with subtyping was almost as good as the whole program polymorphic analysis with subtyping on optimized programs.

Bibliography

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[Aug93] Lennart Augustsson. Implementing haskell overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73, 1993.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI'00*, pages 35–46. ACM Press, June 2000.

[DHM95] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *SAS'95*, September 1995.

[DLFR01] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization, 2001.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *Proc. of SAS'95*, pages 136–153. Springer-Verlag, LNCS 983, September 1995.

[FFA00a] J. Foster, M. Fändrich, and A. Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In *SAS'00*, June 2000.

[FFA00b] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *SAS'00*, pages 175–198, 2000.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In *PLDI'00*, Vancouver B.C., Canada, 2000.

[Ged03] Tobias Gedell. A Case Study on the Scalability of a Constraint Solving Algorithm: Polymorphic Usage Analysis with Subtyping. Master thesis, 2003.

[GS00] Jörgen Gustavsson and Josef Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In Markus Mohnen and Pieter W. M. Koopman, editors, *IFL*, volume 2011 of *Lecture Notes in Computer Science*, pages 140–157. Springer, 2000.

[Gus98] Jörgen Gustavsson. A type based sharing analysis for update avoidance and optimisation. In *ICFP*, pages 39–50. ACM, SIGPLAN Notices 34(1), 1998.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *PLDI'01*, pages 254–263. ACMPress, June 2001.

[JM99] S. Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. In *Workshop on Implementing Declarative Languages*, 1999.

[Jon94] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 107–117, 1994.

[LGH$^+$92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. L. Peyton Jones, and P. Wadler. Avoiding Unnecessary Updates. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, Glasgow, 1992. Springer.

[Mar93] S. Marlow. Update Avoidance Analysis by Abstract Interpretation. In *Proc. 1993 Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer–Verlag, 1993.

[Myc82] Alan Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, University of Edinburg, 1982.

[Myc84] Alan Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *Proceedings 6th International Symposium on Programming, Lecture Notes in Computer Science*, Toulouse, July 1984. Springer Verlag.

[Par93] W. Partain. The nofib benchmark suite of haskell programs, 1993.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. of ICFP'96*, pages 1–12. ACM, SIGPLAN Notices 31(6), May 1996.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *POPL'96*, pages 32–41. ACM Press, January 1996.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. of FPCA*, La Jolla, 1995. ACM Press, ISBN 0-89791-7.

[Wan02] Keith Wansbrough. *Simple Polymorphic Usage Analysis.* PhD thesis, Computer Laboratory, Cambridge University, England, March 2002.

[WPJ99] Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. In *Proc. of POPL'99.* ACM Press, 1999.

[WPJ00] Keith Wansbrough and Simon Peyton Jones. Simple Usage Polymorphism. In *ACM SIGPLAN Workshop on Types in Compilation.* Springer-Verlag, 2000.

A Detailed Results of the Measurements

In three cases the analysis under consideration ran out of memory when analyzing a particular program. For these programs the effectiveness is reported as "-" and is excluded from the computed average.

A.1 Polymorphism

	Effectiveness			
Program	polyrec	monorec	simple-poly	mono
anna	55.61%	54.69%	50.62%	30.64%
bspt	44.98%	44.98%	28.88%	13.98%
cacheprof	38.34%	38.34%	31.76%	12.28%
compress	30.22%	30.22%	18.71%	5.76%
compress2	32.70%	32.70%	20.75%	6.92%
fem	66.55%	66.31%	56.35%	26.50%
fluid	68.84%	68.17%	52.64%	35.47%
fulsom	50.51%	48.48%	37.82%	24.87%
gamteb	59.46%	58.38%	42.16%	22.16%
gg	55.54%	55.25%	45.34%	12.24%
grep	36.02%	36.02%	23.12%	11.29%
hidden	63.92%	63.13%	47.63%	23.58%
hpg	49.51%	45.92%	39.05%	15.20%
infer	48.87%	48.42%	43.02%	18.92%
lift	38.40%	38.02%	34.22%	19.77%
linear	63.90%	63.41%	57.80%	29.76%
maillist	34.90%	34.90%	20.31%	6.25%
mkhprog	46.46%	46.46%	38.19%	9.84%
parser	38.43%	38.43%	34.50%	8.95%
pic	60.71%	59.29%	46.25%	25.00%
polygp	41.86%	41.86%	23.26%	9.30%
prolog	53.42%	53.42%	43.49%	17.81%
reptile	52.64%	50.95%	45.45%	17.97%
rsa	40.36%	38.57%	29.60%	5.83%
rx	65.27%	64.91%	49.82%	32.23%
scs	60.00%	58.69%	47.32%	26.27%
symalg	48.82%	46.75%	38.17%	18.64%
average	49.86%	49.14%	38.75%	18.05%

Figure 5: Polymorphism on unoptimized code

	Effectiveness			
Program	polyrec	monorec	simple-poly	mono
anna	14.74%	13.14%	12.02%	11.86%
bspt	21.43%	21.43%	19.64%	4.46%
cacheprof	12.03%	12.03%	-	10.53%
compress	10.53%	10.53%	0.00%	0.00%
compress2	14.63%	7.32%	0.00%	2.44%
fem	14.89%	14.89%	13.30%	8.51%
fluid	20.28%	20.28%	17.13%	15.38%
fulsom	23.02%	10.32%	7.14%	9.68%
gamteb	4.21%	4.21%	3.16%	2.11%
gg	15.15%	15.15%	12.63%	8.59%
grep	2.63%	2.63%	0.00%	0.00%
hidden	12.50%	10.94%	7.03%	7.03%
hpg	9.57%	9.57%	8.26%	7.83%
infer	2.42%	2.42%	0.00%	0.61%
lift	7.52%	7.52%	6.02%	3.76%
linear	15.56%	15.56%	13.33%	13.33%
maillist	4.76%	4.76%	0.00%	0.00%
mkhprog	1.41%	1.41%	1.41%	1.41%
parser	0.62%	0.62%	0.00%	0.00%
pic	8.62%	8.62%	6.03%	6.03%
polygp	0.00%	0.00%	0.00%	0.00%
prolog	10.45%	10.45%	1.49%	8.96%
reptile	9.00%	9.00%	6.00%	7.00%
rsa	13.33%	13.33%	6.67%	6.67%
rx	26.06%	25.76%	-	14.24%
scs	21.74%	21.74%	17.79%	20.16%
symalg	12.16%	12.16%	6.76%	6.76%
average	11.45%	10.58%	6.63%	6.57%

Figure 6: Polymorphism on optimized code

A.2 Subtyping

	Effectiveness			
Program	polyrec	polyrec-nosub	mono	mono-nosub
anna	55.61%	51.73%	30.64%	2.28%
bspt	44.98%	34.65%	13.98%	2.74%
cacheprof	38.34%	34.62%	12.28%	-
compress	30.22%	23.02%	5.76%	3.60%
compress2	32.70%	26.42%	6.92%	3.14%
fem	66.55%	61.63%	26.50%	5.52%
fluid	68.84%	60.02%	35.47%	5.37%
fulsom	50.51%	42.13%	24.87%	4.57%
gamteb	59.46%	50.63%	22.16%	3.78%
gg	55.54%	48.69%	12.24%	2.92%
grep	36.02%	27.42%	11.29%	4.84%
hidden	63.92%	53.64%	23.58%	1.42%
hpg	49.51%	46.08%	15.20%	3.59%
infer	48.87%	44.37%	18.92%	3.60%
lift	38.40%	34.22%	19.77%	4.56%
linear	63.90%	59.76%	29.76%	4.39%
maillist	34.90%	28.65%	6.25%	3.12%
mkhprog	46.46%	40.55%	9.84%	3.15%
parser	38.43%	36.03%	8.95%	0.22%
pic	60.71%	52.14%	25.00%	3.93%
polygp	41.86%	33.72%	9.30%	2.33%
prolog	53.42%	48.29%	17.81%	4.79%
reptile	52.64%	49.05%	17.97%	3.81%
rsa	40.36%	34.98%	5.83%	3.14%
rx	65.27%	56.07%	32.23%	1.25%
scs	60.00%	52.55%	26.27%	5.23%
symalg	48.82%	44.97%	18.64%	5.33%
average	49.86%	43.56%	18.05%	3.56%

Figure 7: Subtyping on unoptimized code

	Effectiveness			
Program	polyrec	polyrec-nosub	mono	mono-nosub
anna	14.74%	12.50%	11.86%	0.32%
bspt	21.43%	6.25%	4.46%	0.00%
cacheprof	12.03%	3.76%	10.53%	0.00%
compress	10.53%	10.53%	0.00%	0.00%
compress2	14.63%	14.63%	2.44%	0.00%
fem	14.89%	14.36%	8.51%	1.06%
fluid	20.28%	16.08%	15.38%	3.85%
fulsom	23.02%	22.22%	9.68%	2.38%
gamteb	4.21%	4.21%	2.11%	2.11%
gg	15.15%	5.05%	8.59%	1.01%
grep	2.63%	2.63%	0.00%	0.00%
hidden	12.50%	8.59%	7.03%	1.56%
hpg	9.57%	5.65%	7.83%	0.87%
infer	2.42%	1.82%	0.61%	0.00%
lift	7.52%	7.52%	3.76%	0.00%
linear	15.56%	14.44%	13.33%	0.00%
maillist	4.76%	4.76%	0.00%	0.00%
mkhprog	1.41%	0.00%	1.41%	0.00%
parser	0.62%	0.62%	0.00%	0.00%
pic	8.62%	5.17%	6.03%	1.72%
polygp	0.00%	0.00%	0.00%	0.00%
prolog	10.45%	8.96%	8.96%	0.00%
reptile	9.00%	9.00%	7.00%	2.00%
rsa	13.33%	13.33%	6.67%	0.00%
rx	26.06%	19.70%	14.24%	3.03%
scs	21.74%	17.79%	20.16%	0.79%
symalg	12.16%	10.81%	6.76%	4.05%
average	11.45%	8.90%	6.57%	0.92%

Figure 8: Subtyping on optimized code

A.3 Data Types

	Effectiveness				
Program	no limit	100	10	1	0
anna	55.61%	54.81%	47.97%	45.75%	45.44%
bspt	44.98%	43.16%	27.36%	27.63%	20.36%
cacheprof	38.34%	37.72%	31.39%	30.65%	29.40%
compress	30.22%	30.22%	15.83%	15.83%	15.83%
compress2	32.70%	32.70%	15.72%	15.72%	15.72%
fem	66.55%	64.99%	35.37%	34.89%	34.53%
fluid	68.84%	66.73%	40.94%	38.64%	36.63%
fulsom	50.51%	47.21%	33.76%	32.99%	32.49%
gamteb	59.46%	56.76%	26.49%	26.13%	25.59%
gg	55.54%	54.23%	36.73%	36.15%	35.86%
grep	36.02%	35.48%	22.04%	21.51%	21.51%
hidden	63.92%	62.03%	40.03%	38.77%	36.39%
hpg	49.51%	47.22%	37.75%	37.25%	37.25%
infer	48.87%	43.92%	39.64%	39.19%	38.74%
lift	38.40%	38.40%	33.46%	31.94%	31.56%
linear	63.90%	60.49%	39.51%	39.02%	38.54%
maillist	34.90%	33.33%	17.19%	17.19%	17.19%
mkhprog	46.46%	46.06%	37.40%	37.40%	36.61%
parser	38.43%	38.43%	32.53%	32.53%	32.53%
pic	60.71%	56.07%	34.64%	31.61%	30.36%
polygp	41.86%	40.70%	20.93%	20.93%	20.93%
prolog	53.42%	52.40%	41.44%	40.41%	40.41%
reptile	52.64%	52.43%	37.63%	37.00%	37.00%
rsa	40.36%	39.91%	30.04%	30.04%	29.15%
rx	65.27%	64.64%	45.45%	42.50%	39.55%
scs	60.00%	57.39%	37.12%	36.21%	34.12%
symalg	48.82%	47.04%	35.50%	35.21%	35.21%
average	49.86%	48.31%	33.11%	32.34%	31.44%

Figure 9: Data types on unoptimized code

	Effectiveness				
Program	no limit	100	10	1	0
anna	14.74%	14.74%	14.42%	13.46%	13.46%
bspt	21.43%	21.43%	21.43%	21.43%	6.25%
cacheprof	12.03%	12.03%	12.03%	7.52%	4.51%
compress	10.53%	10.53%	10.53%	10.53%	10.53%
compress2	14.63%	14.63%	12.20%	12.20%	12.20%
fem	14.89%	14.89%	14.89%	14.36%	14.36%
fluid	20.28%	20.28%	20.28%	19.23%	16.43%
fulsom	23.02%	23.02%	23.02%	23.02%	22.22%
gamteb	4.21%	4.21%	4.21%	4.21%	4.21%
gg	15.15%	15.15%	14.65%	10.10%	5.05%
grep	2.63%	2.63%	2.63%	2.63%	2.63%
hidden	12.50%	12.50%	12.50%	10.16%	9.38%
hpg	9.57%	9.57%	9.57%	9.13%	9.13%
infer	2.42%	2.42%	2.42%	1.82%	1.82%
lift	7.52%	7.52%	7.52%	7.52%	7.52%
linear	15.56%	15.56%	15.56%	14.44%	14.44%
maillist	4.76%	4.76%	4.76%	4.76%	4.76%
mkhprog	1.41%	1.41%	1.41%	1.41%	1.41%
parser	0.62%	0.62%	0.62%	0.62%	0.62%
pic	8.62%	8.62%	8.62%	8.62%	4.31%
polygp	0.00%	0.00%	0.00%	0.00%	0.00%
prolog	10.45%	10.45%	10.45%	10.45%	8.96%
reptile	9.00%	9.00%	9.00%	8.00%	8.00%
rsa	13.33%	13.33%	13.33%	13.33%	13.33%
rx	26.06%	22.42%	22.12%	16.36%	15.76%
scs	21.74%	21.74%	21.74%	19.76%	18.97%
symalg	12.16%	12.16%	12.16%	10.81%	10.81%
average	11.45%	11.32%	11.19%	10.22%	8.93%

Figure 10: Data types on optimized code

A.4 Whole Program Analysis

	Effectiveness			
Program	polyrec	polyrec-whole	mono	mono-whole
anna	55.61%	59.80%	30.64%	44.27%
bspt	44.98%	48.33%	13.98%	29.79%
cacheprof	38.34%	69.85%	12.28%	25.81%
compress	30.22%	40.29%	5.76%	19.42%
compress2	32.70%	38.39%	6.92%	23.90%
fem	66.55%	69.42%	26.50%	54.92%
fluid	68.84%	73.35%	35.47%	57.62%
fulsom	50.51%	61.68%	24.87%	47.46%
gamteb	59.46%	63.24%	22.16%	48.65%
gg	55.54%	57.73%	12.24%	27.11%
grep	36.02%	43.55%	11.29%	20.97%
hidden	63.92%	71.04%	23.58%	47.15%
hpg	49.51%	54.41%	15.20%	30.23%
infer	48.87%	63.29%	18.92%	35.59%
lift	38.40%	44.11%	19.77%	28.90%
linear	63.90%	71.71%	29.76%	59.02%
maillist	34.90%	45.31%	6.25%	21.88%
mkhprog	46.46%	52.76%	9.84%	18.11%
parser	38.43%	40.39%	8.95%	27.29%
pic	60.71%	66.07%	25.00%	50.71%
polygp	41.86%	52.33%	9.30%	19.77%
prolog	53.42%	60.62%	17.81%	34.59%
reptile	52.64%	57.93%	17.97%	33.83%
rsa	40.36%	44.84%	5.83%	32.74%
rx	65.27%	71.25%	32.23%	57.05%
scs	60.00%	69.54%	26.27%	50.98%
symalg	48.82%	53.55%	18.64%	39.64%
average	49.86%	57.21%	18.05%	36.57%

Figure 11: Whole program analysis on unoptimized code

	Effectiveness			
Program	polyrec	polyrec-whole	mono	mono-whole
anna	14.74%	17.63%	11.86%	17.63%
bspt	21.43%	22.32%	4.46%	22.32%
cacheprof	12.03%	16.54%	10.53%	16.54%
compress	10.53%	21.05%	0.00%	21.05%
compress2	14.63%	24.39%	2.44%	24.39%
fem	14.89%	19.15%	8.51%	18.62%
fluid	20.28%	32.52%	15.38%	24.13%
fulsom	23.02%	35.71%	9.68%	35.71%
gamteb	4.21%	18.95%	2.11%	18.95%
gg	15.15%	18.18%	8.59%	18.18%
grep	2.63%	7.89%	0.00%	7.89%
hidden	12.50%	26.56%	7.03%	23.44%
hpg	9.57%	12.61%	7.83%	11.74%
infer	2.42%	15.76%	0.61%	15.15%
lift	7.52%	16.54%	3.76%	15.79%
linear	15.56%	22.22%	13.33%	20.00%
maillist	4.76%	19.05%	0.00%	19.05%
mkhprog	1.41%	1.41%	1.41%	1.41%
parser	0.62%	1.88%	0.00%	1.88%
pic	8.62%	15.52%	6.03%	15.52%
polygp	0.00%	7.69%	0.00%	7.69%
prolog	10.45%	14.93%	8.96%	14.93%
reptile	9.00%	15.00%	7.00%	15.00%
rsa	13.33%	26.67%	6.67%	26.67%
rx	26.06%	39.39%	14.24%	23.03%
scs	21.74%	32.02%	20.16%	32.02%
symalg	12.16%	16.22%	6.76%	16.22%
average	11.45%	19.18%	6.57%	17.96%

Figure 12: Whole program analysis on optimized code

Embedding Static Analysis into Tableaux and Sequent based Frameworks

Tobias Gedell

Abstract

In this paper we present a method for embedding static analysis into tableaux and sequent based frameworks. In these frameworks, the information flows from the root node to the leaf nodes. We show that the existence of free variables in such frameworks introduces a bi-directional flow, which can be used to collect and synthesize arbitrary information.

We use free variables to embed a static program analysis in a sequent style theorem prover used for verification of Java programs. The analysis we embed is a reaching definitions analysis, which is a common and well-known analysis that shows the potential of our method.

The achieved results are promising and open up for new areas of application of tableaux and sequent based theorem provers.

1 Introduction

The aim of our work is to integrate static program analysis into theorem provers used for program verification. In order to do so, we must bridge the mismatch between the synthetic nature of static program analysis and analytic nature of tableaux and sequent calculi. One of the major differences is the flow of information.

In a program analysis, information is often synthesized by dividing a program into its subcomponents, calculating some information for each component and then merging the calculated information. This gives a flow of information that is directed bottom-up, with the subcomponents at the bottom.

Both tableaux and sequent style provers work in the opposite way. They take a theorem as input and, by applying the rules of their calculi, gradually divide it into branches, corresponding to logical case distinction, until all branches can be proved or refuted. In a ground calculus, there is no flow of information between the branches. Neither is there a need for that since the rules of the calculus only extend the proof by adding new nodes. Because of this, the information flow in a ground calculus is uni-directional—directed top-down, from the root to the leafs of the proof tree.

Tableaux calculi are often extended with *free variables* which are used for handling universal quantification (in the setting of sequent calculi, free variables correspond to *meta variables*, which are used for existential quantification)

[Fit96]. Adding free variables breaks the uni-directional flow of information. When a branch chooses to instantiate a free variable, the instantiation has to be made visible at the point where the free variable was introduced. Therefore, some kind of information flow backwards in the proof has to exist. By exploiting this bi-directional flow we can collect and synthesize arbitrary information which opens up for new areas of application of the calculi.

We embed our program analysis in a sequent calculus using meta variables. The reason for choosing a program analysis is that logics for program verification could greatly benefit from an integration with program analysis. An example of this is the handling of loops in programs. Often a human must manually handle things like loops and recursive functions. Even for program constructs, which a verification system can cope with automatically, the system sometimes performs unnecessary work. Such a system could benefit from having a program analysis that could cheaply identify loops and other program constructs that can be handled using specialized rules of the program logics that do not require user interaction. An advantage of embedding a program analysis in a theorem prover instead of implementing it in an external framework, is that it allows for a closer integration of the analysis and prover.

The main contributions of this work are that:

- We show how synthesis can be performed in a tableau or sequent style prover, which opens up for new areas of application.
- We show how the rules of a program analysis can be embedded into a program logic and coexist with the original rules by using a tactic language.
- We give a proof-of-concept of our method. We do this by giving the full embedding of a program analysis in an interactive theorem prover.

The outline of this paper is as follows: In Section 2 we elaborate more on how we use the bi-directional flow of information; In Section 3 we briefly describe the theorem prover used for implementing our program analysis; In Section 4 we describe the program analysis; in Section 5 we present the embedding of the analysis in the theorem prover; in Section 6 we draw some conclusions; and in Section 7 we discuss future work.

2 Flow of Information

By using the mechanism for free variables we can send information from arbitrary nodes in the proof to nodes closer to the root. This is very useful to us since our program analysis needs to send information from the subcomponents of the program to the root node. In a proof, the subcomponents of the program correspond to leaf nodes. To show how it works, consider a tableau created with a destructive calculus where, at the root node, a free variable I is introduced. When I is instantiated by a branch closure, the closing substitution is applied to all branches where I occurs. This allows us to embed various analyses. One could, for example, imagine a very simple analysis that finds out whether a property P is true for any branch in a proof. In order to do so, we modify the closure rule. Normally, the closure rule tries to find two formulas φ and $\neg\psi$ in the same branch and a substitution that unifies φ and ψ. The new closure rule is modified to search for a closing substitution for a branch and if it finds one, check whether P is true for the particular branch. If it is, then the closing substitution is extended with an instantiation of the free variable I to a constant symbol c. We can now use this calculus to construct a proof as usual and when it is done, check whether I has been instantiated or not. If it has, then we know that P was true for at least one of the branches. Note that we are not interested in *what* I was instantiated to, just the fact that it *was* instantiated.

There is still a limit to how much information that can be passed to the root node. It is not possible to gather different information from each branch closure since they all use the same variable, I, to send their information. In particular, the reaching definitions analysis that we want to embed needs to be able to compute different information for each branch in the proof.

This can be changed by modifying the extension rule. When two branches are created in a proof, two new free variables, I_L and I_R, are introduced and I instantiated to $\mathit{branch}(I_L, I_R)$. I_L is used as the new I-variable for the left branch and I_R for the right branch. By doing this we ensure that each branch has its own variable for sending information. This removes the possibility of conflicting instantiations, since each I-variable will be instantiated at most once, either by extending or closing the branch to which it belongs.

When the tableau shown in Figure 1 has been closed, we get the instantiation of I, which will be the term $\mathit{branch}(\mathit{branch}(\mathit{info}_1, \mathit{info}_2), \mathit{branch}(\mathit{info}_3, \mathit{info}_4))$ that contains the information collected from all four branches. We have now used the tableau calculus to synthesize information from the leaf nodes.

We are now close to being able to implement our program analysis. The remaining problem is that we want to be able to distinguish between different types of branches. An example of this is found in Section 4.2 where different types of branches compute different collections of equations. We overcome this problem by, instead of always using the function symbol *branch*, allowing arbitrary function symbols when branching.

2.1 Non Destructive Calculi

In a non destructive constraint tableau, as described in [Gie01], it is possible to embed analyses using the same method.

In a constraint tableau, each node n has a *sink* object that contains all closing substitutions for the sub tableau having n as its top node. When adding a node to a branch, all closing substitutions of the branch are added to the node's sink object. The substitutions in the sink object are then sent to the sink object of the parent node. If the parent node is a node with more than one child, it has a *merger* object that receives the substitution and checks whether it is a closing substitution for all children. If it is, then it is propagated upwards to the sink object of the parent node, otherwise it is discarded. If the parent

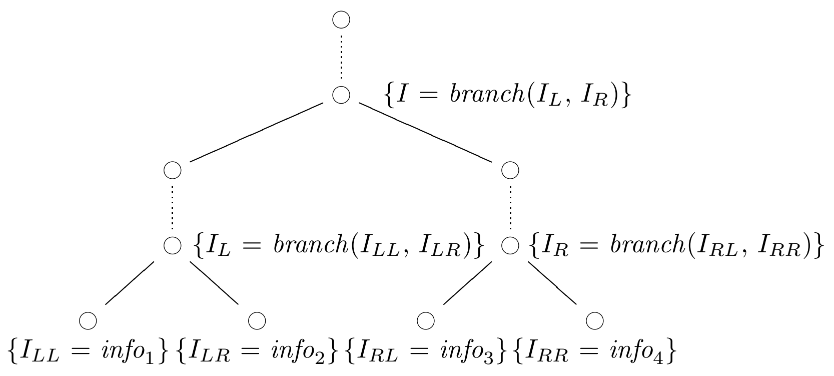
$\{I = branch(I_L, I_R)\}$
$\{I_L = branch(I_{LL}, I_{LR})\}$
$\{I_R = branch(I_{RL}, I_{RR})\}$
$\{I_{LL} = info_1\}$
$\{I_{LR} = info_2\}$
$\{I_{RL} = info_3\}$
$\{I_{RR} = info_4\}$

Figure 1: Example tableau

node only has one child, the substitution is directly sent to the node's parent node.

A tableau working like this is called non destructive since the free variables are never instantiated. Instead, a set of all possible closing instantiations is calculated for each branch and propagated upwards. When a closing substitution reaches the root node, the search is over since we know that it closes the entire tableau.

Using our method in a non destructive constraint tableau is easy. We modify the sink object of the root node to not only, when a closing substitution is found, tell us that the tableau is closable but also give us the closing substitution. The infrastructure with the sink objects could also make it easy to implement some of the extensions described in Section 7.

3 The KeY Prover

For the implementation, we choose an interactive theorem prover with a tactic programming language, the KeY system [ABB+05]. The KeY system is a theorem prover for the Java Card language that uses a dynamic logic [Bec01]. The dynamic logic is a modal logic in which Java programs can occur as parts of formulas. An example of this is the formula,

```
<{ i = 1; }> i > 0  ,
```

that denotes that after executing the assignment `i = 1;` the value of the variable i is greater than 0.

The KeY system is based on a non destructive sequent calculus with a standard semantics. It is well known that sequent calculi can be seen as the duality of tableaux calculi and we use this to carry over the method described in Section 2 to the sequent calculus used by KeY.

3.1 Tactic Programming Language

Theorem provers for program verification typically need to have a large set of rules at hand to handle all constructs in a language. Instead of hard-wiring these into the core of the theorem prover, one can opt for a more general solution and create a domain specific tactic language, which is used to implement the rules.

The rules written in the tactic language of KeY are called taclets [BGH+04]. A taclet can be be seen as an implementation of a sequent calculus rule. In most theorem provers for sequent calculi, the rules perform some kind of pattern matching on sequents. Typically, the rules consist of a guard pattern and an action. If a sequent matches the guard pattern then the rule is applied and the action performed on the sequent. What it means for the pattern of a taclet to match a sequent is that there is a unifying substitution for the pattern and the sequent under consideration. The actions that can be performed include closing a proof branch, creating modified copies of sequents, and creating new proof branches.

We now have a look at the syntax of the tactic language and start with one of the simplest rules, the `close_by_true` rule.

```
close_by_true {
 find (==> true)
 close goal
};
```

The pattern matches sequents where *true* can be found on the right hand side. If *true* can be found on the right hand side, we know that we can close the proof branch under consideration, which is done by the `close goal` action.

If we, instead of closing the branch, want to create a modified copy of the sequent we use the `replacewith` action.

```
not_left {
 find (!b ==>)
 replacewith (==> b)
};
```

If we find a negated formula b on the left hand side we replace it with b on the right hand side.[1] The proof branch will remain open, but contain the modified sequent. We can also create new proof branches by using multiple `replacewith` actions.

So far, we have only considered sequents that do not contain embedded Java programs. When attaching programs to formulas, one has to choose a modality operator. There are a number of different modality operators having different semantics. The diamond operator `<{p}>`ϕ says that there is a terminating execution of the program `p` after which the formula ϕ holds. The box operator `[{p}]`ϕ says that after all terminating executions the formula ϕ holds. For our purpose, the modalities do not have any meaning since we are not trying to

[1] Note that Γ and Δ are only implicitly present in the taclet.

construct a proof in the traditional way. Regardless of this, the syntax of the taclet language forces us to have a modality operator attached to all programs. We, therefore, arbitrarily choose to use the diamond operator. In the future, it would be better to have a general-purpose operator with a free semantics that could be used for cases like this.

As an example of a taclet matching an embedded Java program, consider the following taclet, that matches an assignment of a literal to a variable attached to the formula *true* and closes the proof branch:

```
term_assign_literal {
 find (==> <{#var = #literal;}>(true))
 close goal
};
```

4 Reaching Definitions Analysis

The analysis we choose to implement using our technique is *reaching definitions analysis* [NNH99]. This analysis is commonly used by compilers to perform several kinds of optimization such as, for example, loop optimization and constant computation [ASU86]. The analysis calculates which assignments may reach each individual statement in a program. Consider the following program, consisting of three assignments, where each statement is annotated with a label so that we can uniquely identify them.

$$\texttt{a} \overset{0}{=} \texttt{1; b} \overset{1}{=} \texttt{1; a} \overset{2}{=} \texttt{1;}$$

Let us look at the statement annotated with 1. The statement executed before it (which we will call its previous statement) is the assignment $\texttt{a} \overset{0}{=} \texttt{1;}$ and since a has not yet been reassigned it still contains the value 1. We say that the assignment annotated with 0 *reaches* the statement annotated with 1. For each statement, we calculate the set of labels of the assignments that reach the statement before and after it has been executed. We call these sets the entry and exit sets. For this example, the label 0 will be in the entry set of the last assignment but not in its exit set, since the variable a is re-assigned. We do not just store the labels of the assignments in the sets, but also the name of the variable that is assigned. The complete entry and exit sets for our example program look as follows:

label	**Entry**	**Exit**
0	{}	{(a, 0)}
1	{(a, 0)}	{(a, 0), (b, 1)}
2	{(a, 0), (b, 1)}	{(b, 1), (a, 2)}

It is important to understand that the results of the analysis will be an *approximation*. It is undecidable to calculate the exact reaching information, which can easily be proven by using the halting problem. We will, however, ensure that the approximation is *safe*, which in this context means that if an assignment reaches a statement then the label of the assignment must be present in the entry set of that statement. The reverse may not hold, a label of an assignment being present in an entry set of a statement, does not necessarily mean that the assignment may reach that statement.

It is easy to see that for any program, a sound result of the analysis would be to let all entry and exit sets be equal to the set of all labels occurring in the program. This result would, of course, not be useful; what we want are as precise results as possible.

The analysis consists of two parts: a constraint-generation part and a constraint-solving part. The constraint-generation part traverses the program and generates a collection of equations defining the entry and exit sets. The equations are then solved by the constraint-solving part that calculates the actual sets.

4.1 Input Language

As input language, we choose a very simple language, the WHILE-language, which consists of assignments, block statements and if- and while-statements. We choose a simple language because we do not want to wrestle with a large language but instead show the concept of how the static program analysis can be implemented.

In the language, a program consists of a number of statements.

$$
\begin{array}{llcl}
\text{Programs} & \mathit{program} & ::= & \mathit{stmt}^{+} \\
\text{Statements} & \mathit{stmt} & ::= & \mathit{var} \overset{\mathit{lbl}}{=} \mathit{expr}\mathbf{;} \\
 & & | & \mathbf{if}_{\mathit{lbl}}\mathbf{(}\mathit{term}\mathbf{)}\ \mathit{stmt}\ \mathbf{else}\ \mathit{stmt} \\
 & & | & \mathbf{while}_{\mathit{lbl}}\mathbf{(}\mathit{term}\mathbf{)}\ \mathit{stmt} \\
 & & | & \mathbf{\{}\mathit{stmt}^{*}\mathbf{\}}
\end{array}
$$

lbl ranges over the natural numbers and will be unique for each statement. We do not annotate block statements since they are just used to group multiple statements.

To simplify our analysis, we impose the restriction that all expressions *expr* must be free from side-effects. Since removing side-effects from expressions is a simple and common program transformation, this restriction is reasonable to make.

4.2 Rules of the Analysis

We now look at the constraint-generation part of the analysis and start by defining the collections of equations that will be generated. These equations will characterize the reaching information in the analyzed program.

$$\begin{array}{llll} \text{Equations} & \Pi & ::= & \emptyset \\ & & | & \mathbf{Entry}(\mathit{lbl}) = \Sigma \\ & & | & \mathbf{Exit}(\mathit{lbl}) = \Sigma \\ & & | & \Pi \wedge \Pi \end{array} \tag{1}$$

$\emptyset$ is the empty collection of equations. $\mathbf{Entry}(\mathit{lbl}) = \Sigma$ and $\mathbf{Exit}(\mathit{lbl}) = \Sigma$ are equations defining the entry and exit sets of the statement annotated with *lbl* to be equal to the set expression Σ. We let $\wedge$ be the conjunction operator that merges two collections of equations.

The set expressions,

$$\begin{array}{llll} \text{Set expressions} & \Sigma & ::= & \emptyset \\ & & | & (\mathit{var}, \mathit{lbl}) \\ & & | & \mathbf{Entry}(\mathit{lbl}) \\ & & | & \mathbf{Exit}(\mathit{lbl}) \\ & & | & \Sigma \cup \Sigma \\ & & | & \Sigma \text{-} \Sigma \end{array} , \tag{2}$$

are used to build up the entry and exit sets. $\emptyset$ is the empty set (the overloading of this symbol will not cause any confusion). $(\mathit{var}, \mathit{lbl})$ is the set consisting of only a single reaching assignment. $\mathbf{Entry}(\mathit{lbl})$ and $\mathbf{Exit}(\mathit{lbl})$ refer to the values of the entry and exit sets of the statement annotated with *lbl*. $\cup$ and **-** are the union and difference operators.

The rules of the analysis are of the form $\ell_0 \vdash s \Downarrow \ell_1 : \Pi$, where s is the statement under consideration, ℓ_0 is the label of the statement executed before s (we will sometimes call this statement the *previous* statement), ℓ_1 the label of the last executed statement in s, and Π the equations characterizing the reaching information of the statement s.

The intuition behind this form is that we need to know the label of the statement executed before s because we will use its exit set when analyzing s. After we have analyzed s, we need to know the label of the last executed statement in s (which will often be s itself) because the statement executed after s needs to use the right exit set. Then, the most important thing to know is, of course, what equations were collected when analyzing s.

In the assignment rule,

$$\frac{\textsc{Assign}}{\begin{array}{l}\ell_0 \vdash x \overset{\ell_1}{=} e\mathbf{;} \Downarrow \ell_1 : \mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0) \wedge \\ \mathbf{Exit}(\ell_1) = (x, \ell_1) \cup (\mathbf{Entry}(\ell_1) - \bigcup_{\ell \in \mathit{lbl}} (x, \ell))\end{array}} ,$$

we know that the reaching assignments in the entry set will be exactly those that were reaching after the previous statement was executed. This is expressed by the equation $\mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0)$. For the exit set, we know that all previous assignments of x will no longer be reaching. The assignments of all other variables will remain untouched. We therefore let the exit set be equal to the entry set from which we have first removed all previous assignments of x and then added the assignment (x, ℓ_1). This is expressed by the equation $\mathbf{Exit}(\ell_1) = (x, \ell_1) \cup (\mathbf{Entry}(\ell_1) \; - \bigcup_{\ell \in lbl}(x, \ell))$.

So far, we have not seen the need for including the label of the previous statement in the rules. This is illustrated by the rule for if-statements:

If

$$\frac{\ell_0 \vdash s_0 \;\Downarrow\; \ell_2 \;:\; \Pi_0 \qquad \ell_0 \vdash s_1 \;\Downarrow\; \ell_3 \;:\; \Pi_1}{\begin{array}{c}\ell_0 \vdash \mathbf{if}_{\ell_1}(e)\; s_0 \;\mathbf{else}\; s_1 \;\Downarrow\; \ell_1 \;:\; \Pi_0 \;\wedge\; \Pi_1 \;\wedge\; \mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0) \;\wedge \\ \mathbf{Exit}(\ell_1) = \mathbf{Exit}(\ell_2) \cup \mathbf{Exit}(\ell_3)\end{array}}$$

For an if-statement, the entry set will be equal to the exit set of the previous statement, which is expressed by the equation $\mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0)$. When analyzing the two branches s_0 and s_1, we use l_0 as the label of the previous statement since it is important that they, when referring to the exit set of the previous statement, use $\mathbf{Exit}(l_0)$ and not the exit set of the if-statement. From the two branches, we get the collections of the generated equations Π_0 and Π_1, along with the labels l_2 and l_3, which are the labels of the last executed statements in s_0 and s_1. Since we do not know which branch is going to be taken, we must approximate and assume that both branches can be taken. The exit set of the if-statement will therefore be equal to the union of the exit set of the last executed statements in s_0 and s_1, expressed by the equation $\mathbf{Exit}(\ell_1) = \mathbf{Exit}(\ell_2) \cup \mathbf{Exit}(\ell_3)$.

The rule for while-statements,

While

$$\frac{\ell_1 \vdash s \;\Downarrow\; \ell_2 \;:\; \Pi_0}{\begin{array}{c}\ell_0 \vdash \mathbf{while}_{\ell_1}(e)\; s \;\Downarrow\; \ell_1 \;:\; \Pi_0 \;\wedge\; \mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0) \cup \mathbf{Exit}(\ell_2) \;\wedge \\ \mathbf{Exit}(\ell_1) = \mathbf{Entry}(\ell_1)\end{array}} \quad ,$$

differs significantly from the rule for if-statements. For the entry set, we include the exit set of the last executed statement before the loop, but also the exit set of the last executed statement in the loop body. We must do this because there are two execution paths leading to the while loop. The first is from the statement executed before the loop, and the second from executing the loop body. For the exit set, we do not know if the body was executed or not. We could, therefore, let the exit set be equal to the union of the entry set of the while-statement and the exit set of the last executed statement in s. Since this is exactly what the entry set is defined to be, we just let the exit set be equal to the entry set. When analyzing the body of the loop we must once again approximate. The first time s is executed, it should use the exit set of l_0, since that was the last statement executed. The second time and all times after that, it should instead use the exit set of l_1, since the body of the while loop was the last statement executed. We approximate this by not separating the two cases and always use l_1 as the label of the previous statement.

We do not have a special rule for programs. Instead, we treat a program as a block statement and use the rules for sequential statements, which should not require much description:

Seq-Empty
$$\frac{}{\ell_0 \vdash \mathbf{\{\}} \ \Downarrow\ \ell_0 \ :\ \emptyset}$$

Seq
$$\frac{\ell_0 \vdash s_1 \ \Downarrow\ \ell_1 \ :\ \Pi_1 \quad \cdots \quad \ell_{n-1} \vdash s_n \ \Downarrow\ \ell_n \ :\ \Pi_n}{\ell_0 \vdash \mathbf{\{} s_1 \ \ldots\ s_n \mathbf{\}} \ \Downarrow\ \ell_n \ :\ \Pi_1 \wedge \cdots \wedge \Pi_n}$$

5 Embedding the Analysis into the Prover

5.1 Encoding the Datatypes

In order to encode Σ, Π, and labels, we must declare the types we want to use. We declare **VarSet** which is the type of Σ, **Equations** which is the type of Π and **Label** which is the type of labels. The type of variable names, **Quoted**, is already defined by the system.

In the constructors for Σ, defined by (2), we have, for convenience, replaced the difference operator with the constructor **CutVar**. **CutVar**(s, x) denotes the set expression $s - \bigcup_{\ell \in lbl}(x, \ell)$. Our constructors are defined as function symbols by the following code:

```
VarSet Empty;
VarSet Singleton(Quoted, Label);
VarSet Entry(Label);
VarSet Exit(Label);
VarSet Union(VarSet, VarSet);
VarSet CutVar(VarSet, Quoted);
```

The constructors for Π, defined by (1), are defined analogously to the ones for Σ:

```
Equations None;
Equations EntryEq(Label, VarSet);
Equations ExitEq(Label, VarSet);
Equations Join(Equations, Equations);
```

The KeY system does not feature a unique labeling of statements so we need to annotate each statement ourselves. In order to generate the labels we define the **Zero** and **Succ** constructors with which we can easily enumerate all needed labels. The first label will be **Zero**, the second **Succ(Zero)**, the third **Succ(Succ(Zero))**, and so on.

```
Label Zero;
Label Succ(Label);
```

Since the rules of the analysis refer back to the exit set of the previous statement, there is a problem with handling the very first statement of a program (which does not have any previous statement). To solve this problem we define the label **Start** which we exclusively use as the label of the (non-existing) statement before the first statement. When solving the equations we let the exit set of this label, **Exit(Start)**, be the empty set.

```
Label Start;
```

Since one can only attach *formulas* to embedded Java programs, we need to wrap our parameters in a predicate. The parameters we need are exactly those used in our judgments,

$$\ell_0 \vdash s \Downarrow \ell_1 : \Pi \quad .$$

We wrap the label of the previous statement, ℓ_0, the label of the last executed statement, ℓ_1, and the collection of equations, Π, in a predicate called *wrapper* (we do not need to include the statement s since the wrapper will be attached to it). In the predicate, we also include two labels needed for the generation of the labels used for annotating the program: the first unused label before annotating the statement and the first unused label after annotated the statement. The wrapper formula looks as follows:

```
wrapper(Label, Label, Equations, Label, Label);
```

5.2 Encoding the Rules

Before implementing the rules of our analysis as taclets, we declare the variables that we want to use in our taclets. These declarations should be fairly self explanatory.

```
program variable #x;
program simple expression #e;
program statement #s, #s0, #s1;
Equations pi0, pi1, pi2;
Label lbl0, lbl1, lbl2, lbl3, lbl4, lbl5;
Quoted name;
```

We now look at how the rules of the analysis are implemented and start with the rule for empty block statements. When implemented as a taclet we let it match an empty block statement, written as `<{ {} }>`, and a wrapper formula where the first argument is equal to the second argument, the collection of equations is empty, and the fourth argument is equal to the fifth. The formula pattern is written as `wrapper(lbl0, lbl0, None, lbl1, lbl1)`. The action that should be performed when this rule is applied is that the current proof branch should be closed. This is the case because the Seq-Empty rule has no premises. The complete taclet is written as follows:

```
rdef_seq_empty {
 find (==> <{{}}>(wrapper(lbl0, lbl0, None, lbl1, lbl1)))
 close goal
};
```

The rule for non-empty block statements is a bit more tricky. The rule handles an arbitrary number of statements in a block statement. This is, however, hard to express in the taclet language. Instead, we modify the rule to separate the statements into the head and the trailing list. This is equivalent to the original rule except that a block statement needs one application of the rule for each statement it contains. After being modified, the rule looks like this, where we let $\bar{s_2}$ range over lists of statements:

$$\frac{\ell_0 \vdash s_1 \ \Downarrow\ \ell_1 \ :\ \Pi_1 \qquad \ell_1 \vdash \{\bar{s_2}\} \ \Downarrow\ \ell_2 \ :\ \Pi_2}{\ell_0 \vdash \{s_1\ \bar{s_2}\} \ \Downarrow\ \ell_2 \ :\ \Pi_1 \ \wedge\ \Pi_2}\ \textsc{Seq-modified}$$

When implemented as a taclet, we let it match the head and the tail of the list, written as `<{.. #s1 ...}>`. In this pattern, `#s1` matches the head and the dots, `.. ...`,[2] match the tail. We also let it match a wrapper formula containing the necessary labels together with the conjunction of the two collections of equations Π_1 and Π_2. For each premise, we create a proof branch by using the `replacewith` action. Note how the two last labels are threaded through the taclet:

```
rdef_seq {
 find (==> <{.. #s1 ...}>(wrapper(lbl0, lbl2, Join(pi1, pi2),lbl3,lbl5)))
 replacewith (==> <{#s1}>(wrapper(lbl0, lbl1, pi1, lbl3, lbl4)));
 replacewith (==> <{.. ...}>(wrapper(lbl1, lbl2, pi2, lbl4, lbl5)))
};
```

In the rule for assignments, we must take care of the annotation of the assignment. Since we know that the fourth argument in the wrapper predicate is the first free label, we bind `lbl1` to it. We then use `lbl1` to annotate the assignment. Since we have now used that label, we must increment the counter of the first free label. We do that by letting the fifth argument be the successor of `lbl1`. (Remember that the fifth argument in the wrapper predicate is the first free label after annotated the statement.) In the taclet we use a **varcond** construction to bind the name of the variable matching `#x` to `name`.

```
rdef_assign  {
 find (==> <{#x = #e;}>
  (wrapper(lbl0, lbl1,
          Join(EntryEq(lbl1, Exit(lbl0)),
               ExitEq (lbl1, Union(Singleton(name, lbl1),
                                   CutVar(Entry(lbl1), name)))),
          lbl1, Succ(lbl1))))
 varcond (name quotes #x)
 close goal
};
```

[2]The leading two dots match the surrounding context which for our analysis is known to always be empty. They are however still required by the KeY system.

The taclet for if-statements is larger than the previously shown taclets, but since it introduces no new concepts, it should be easily understood:

```
rdef_if {
 find (==> <{if(#e) #s0 else #s1}>
  (wrapper(lbl0, lbl1,
           Join(Join(pi0, pi1),
                Join(EntryEq(lbl1, Exit(lbl0)),
                     ExitEq (lbl1, Union(Exit(lbl2), Exit(lbl3))))),
           lbl1, lbl5)))
  replacewith (==> <{#s0}>(wrapper(lbl0, lbl2, pi0, Succ(lbl1), lbl4)));
  replacewith (==> <{#s1}>(wrapper(lbl0, lbl3, pi1, lbl4, lbl5)))
};
```

This is also the case with the taclet for while-statements and it is, therefore, left without further description:

```
rdef_while {
 find (==> <{while(#e) #s}>
 (wrapper(lbl0, lbl1,
          Join(pi0, Join(EntryEq(lbl1, Union(Exit(lbl0),Exit(lbl2))),
                         ExitEq (lbl1, Entry(lbl1)))),
          lbl1, lbl3)))
  replacewith (==> <{#s}>(wrapper(lbl1, lbl2, pi0, Succ(lbl1), lbl3)))
};
```

5.3 Experiments

We have tested the implementation of our analysis on a number of different programs. For all tested programs the analysis gave the expected entry and exit sets, which is not that surprising since there is a one-to-one correspondence between the rules of the analysis and the taclets implementing them.

As an example, consider the minimal program `a = 1;`, consisting of only an assignment. We embed this program in a formula, over which we existentially quantify the equations, `s`, the label of the last executed statement, `lbl0`, and the first free label after annotated the program, `lbl1`:

```
ex lbl0:Label. ex s:Equations. ex lbl1:Label.
 <{ a = 1; }>wrapper(Start, lbl0, s, Zero, lbl1)
```

When applying the rules of the analysis, the first thing that happens in that `lbl0`, `s`, and `lbl1` are instantiated with meta variables. This is done by a built-in rule for existential quantification. The resulting formula is the following where `L0`, `S` and `L1` are meta variables:

```
<{ a = 1; }>wrapper(Start, L0, S, Zero, L1)
```

We know that the KeY system will succeed in automatically applying the rules since the analysis is complete and, therefore, works for all programs. Being complete is an essential property for all program analyses and for our analysis it is easy to see that for any program there exists a set of equations which characterize the reaching information of the program.

When the proof has been created, we fetch the instantiation of all meta variables, which for our example are the following.

```
{
 S : Equations =
    Join(
     EntryEq(L0, Exit(Start)),
     ExitEq (L0, Union(Singleton(a, L0), CutVar(Entry(L0), a)))),
 L0 : Label = Zero,
 L1 : Label = Succ(L0)
}
```

We take these constraints and let a stand-alone constraint solver solve them. Recall that the analysis is divided into two parts. The first part, which is done by the KeY system, is to collect the constraints. The second part, which is done by the constraint solver, solves the constraints.

The constraint solver extracts the equations from the constraints and solves them yielding the following sets, which is the expected result:

```
Entry_0 = {}
Exit_0  = {(a, 0)}
```

6 Conclusions

It is interesting to see how well-suited an interactive theorem prover such as the KeY system is to embed the reaching definitions analysis in. One reason for this is that the rules of the dynamic logic are, in a way, not that different from the rules of the analysis. They are both syntax-driven, i.e., which rule to apply is decided by looking at the syntactic shape of the current formula or statement. This shows that theorem provers with free variables or meta variables can be seen as not just theorem provers for a specific logic but, rather, as generic frameworks for syntactic manipulation of formulas. Having this view, it is not that strange that we can be rather radical and disregard the usual semantic meaning of the tactic language, and use it for whatever purpose we want.

The key feature that allows us to implement our analysis is the machinery for meta variables, that we use to create a bi-directional flow of information. Using meta variables, we can let our analysis collect almost any type of information. We are, however, limited in what calculation we can do on the information. So far, we cannot do any calculation on the information while constructing the proof. We cannot, for example, do any simplification of the set expressions. One possible way of overcoming this would be to extend the constraint language to not just include syntactic constraints but also semantic constraints.

When it comes to the efficiency of the implementation of the constraint-generation part, it is a somewhat open issue. One can informally argue that the overhead of using the KeY system, instead of writing a specialized tool for the analysis, should be a constant factor. It might be the case that one needs to optimize the constraint solver to handle unification constraints in a way that is more efficient for the analysis. An optimized constraint solver should be able to handle all constraints, generated by the analysis, in a linear way.

7 Future Work

This work presented in this paper is a starting point and opens up for a lot of future work:

- Try different theorem provers to see how well the method presented in this paper works for other theorem provers.
- Further analyse the overhead of using a theorem prover to implement program analyses.
- Modify the calculus of the KeY prover to make use of the information calculated by the program analysis. We need to identify where the result of the analysis can help and how the rules of the calculus should be modified to use it. It is when this is done that the true potential of the integration is unleashed.
- Explore other analyses. We chose to implement the *reaching definitions analysis* because it is a well known and simple analysis that is well suited for illustrating our ideas. Now that we have shown that it is possible to implement a static program analysis in the KeY system, it is time to look for the analyses that would benefit the KeY system the most. Among the possible candidates for this are:
  - An analysis that calculates the possible side-effects of a method. For example what objects and variables that may change.
  - A path-based flow analysis helping the KeY system to resolve aliasing problems.
  - A flow analysis calculating the set of possible implementation classes of objects. This would help reducing the branching for abstract types like interfaces and abstract classes
  - A null pointer analysis that identifies object references which are not equal to null. This would help the system which currently has to always check whether a reference is equal to null before using it.

One limitation of the sequent calculus in the KeY prover is that the unification constraints, used for instantiating the meta variables, can only express syntactic equality. This is a limitation since it prevents the system from doing any semantic simplification of the synthesized information. If it was able to perform simplification of the information while it is synthesized, not only could it make the whole process more efficient, but also let it guide the construction of the proof to a larger extent. Useful extensions of the constraint language are for example the common set operations: test for membership, union, intersection, difference and so on. In a constraint tableaux setting, the simplification of these operations would then take place in the sink objects associated with each node in the proof.

A more general issue that is not just specific to the work presented in this paper is on which level static program analysis and theorem proving should be integrated. The level of integration can vary from having a program analysis run on a program and then give the result of the analysis together with the program to a theorem prover, to having a general framework in which program analysis and theorem proving are woven together. The former kind of integration is no doubt the easiest to implement but also the most limited. The latter is much more dynamic and allows for an incremental exchange of information between the calculus of the prover and program analysis.

Acknowledgments

We would like to thank Wolfgang Ahrendt, Martin Giese and Reiner Hähnle for the fruitful discussions and help with the KeY system. We thank the following people for reading the draft and providing valuable feedback: Richard Bubel, Jörgen Gustavsson, Kyle Ross and Philipp Rümmer.

Bibliography

[ABB$^+$05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[BGH$^+$04] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: a new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie*

A: Matemáticas, to appear, 2004. Special Issue on Computational Logic.

[Fit96] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, second edition, 1996.

[Gie01] Martin Giese. Incremental Closure of Free Variable Tableaux. In *Proc. Intl. Joint Conf. on Automated Reasoning, Siena, Italy*, number 2083 in LNCS, pages 545–560. Springer-Verlag, 2001.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

Automating Verification of Loops by Parallelization

Tobias Gedell Reiner Hähnle

Abstract

Loops are a major bottleneck in formal software verification, because they generally require user interaction: typically, induction hypotheses or invariants must be found or modified by hand. This involves expert knowledge of the underlying calculus and proof engine. We show that one can replace interactive proof techniques, such as induction, with automated first-order reasoning in order to deal with parallelizable loops, where a loop can be parallelized whenever it avoids dependence of the loop iterations from each other. We develop a dependence analysis that ensures parallelizability. It guarantees soundness of a proof rule that transforms a loop into a universally quantified update of the state change information represented by the loop body. This makes it possible to use automatic first order reasoning techniques to deal with loops. The method has been implemented in the KeY verification tool. We evaluated it with representative case studies from the Java Card domain.

1 Introduction

It is generally agreed that loops and recursive calls are the main bottleneck in formal software verification. The source of the problem is that loops and recursion are proof theoretically handled either with invariant rules or with induction. In both cases, it is necessary in general to strengthen invariants and induction hypotheses in order to make proofs go through. There are also many technicalities with those rules that make their application difficult. A number of heuristic techniques have been developed to guide induction proofs and to find appropriate induction hypotheses (for example, [BM88, BBHI05]).

The context of the present work is formal verification of functional properties of sequential Java programs [ABB$^+$05]. Here the situation is aggravated by the fact that the above mentioned techniques have been developed for relatively simple functional programming languages and are not readily applicable to a complex, imperative, object-based language such as Java (similar comments apply to C, C++, or C#). Hence, not only is there a lack of heuristic techniques that help to automate proofs about loops in Java, but due to the complexity of loop rules in imperative languages [BSS05] user interaction involves a high amount of technical knowledge and is extremely expensive.

A recent divide-and-conquer technique for decomposition of induction proofs [OW05] works for imperative programs, but it is targeted at simplifying user interaction rather than eliminating it. In order to deal *automatically* with loops in verification of JAVA-like languages there are not many options at present: abstraction [Hol02] and approximation [FLL$^+$02] are incomplete and in some scenarios even unsound. They impose also limits on what can be expressed in specifications. If the number of loop iterations is known and small then it is possible to use symbolic execution with finite unwinding [HM05]. The state of the art in JAVA verification is, however, that complex user interaction is unavoidable for almost all loops [Bre06].

In this paper we present an *automatic* deductive verification technique that is applicable to many loops occurring in practically relevant JAVA programs. Like any automatic method it cannot handle all loops, but it is seamlessly integrated with a complete interactive verification system. In addition, it computes useful information even when it fails. To make things concrete, we look at an example (where e(`i`) is an expression with an occurrence of `i`):

```
for (int i = 0; i < a.length; i++) a[i] = e(i);
```

The effect of this piece of code is simply to initialize all elements of the array `a` with the expression e(`i`) at index `i`. Since the length of `a` is in general unknown, it is not possible to deal with this loop by finite unwinding. An abstraction of this program has difficulties to record that the value `a.length` depends on `a`. On the other hand, in most cases it is overkill to use induction on such a simple problem. In order to describe the effect of such loops it is usually sufficient to be able to quantify universally over state update expressions that are performed in parallel. From a proof theoretic point of view, quantified state modifiers can be handled by skolemization and simplification [RÖ6], hence, they are amenable to automated proof search.

In general, the initialization, guard and step expressions, as well as the loop body could be more complicated than in the example above. We are looking for a technique that does not rely on the target program being in a particular syntactic form. Of course, we need to make certain assumptions to ensure that the effect of a loop is expressable as a quantified update. This problem is closely related to loop vectorization and parallelization and it is possible to use notions developed in these fields. The main issue is to exclude certain data dependencies. For example, in the case of e(`i`) $\equiv$ `a[i - 1]` the code above cannot be transformed into a quantified state update, because the updates for each `i` cannot be performed in parallel.

The contribution of this paper is a deductive verification method for treating loops[1]based on the ideas just sketched. Its main properties are:

Robustness The target program needs not to be in a particular syntactic form.
This is achieved by computing the accumulated effect of the expressions

[1]The technique is applicable both to for- and while-loops. In this presentation we concentrate on the former to make the presentation more concise, and because for-loops are much more common in our application domain JAVA CARD.

and statements occurring in the loop by symbolic execution *before* checking the dependencies in the loop body (Section 4 and Section 5).

Soundness There is an automatic dependence analysis that guarantees sound applicability (Section 6).

Automation Proof theoretic treatment of the effect of loops is not by induction but by universally quantified state modification and is automatic (Section 7).

Relevance The method applies not only to a few academic examples, but to a substantial number of loops in realistic programs. An experimental evaluation of a number of realistic Java Card programs confirms this (Section 9).

In the following section, we collect a number of technical notions needed later on. Then, in Section 3, we walk informally through the method guided by an example. The remaining sections then give the technical details.

2 Basic Definitions

The platform for our experiments is the KeY tool [ABB$^+$05], which features an interactive theorem prover for formal verification of sequential Java programs.

2.1 Dynamic Logic for Java Card

In KeY the target program to be verified and its specification are both modeled in an instance of a dynamic logic (DL) [HKT00] calculus called Java DL [Bec01]. Java DL extends other variants of DL used for theoretical investigations or verification purposes, because it handles such phenomena as side effects, aliasing, object types, exceptions, and finite integer types. Java DL axiomatizes full Java minus multi-threading, floating point types, and dynamic class loading.

Deduction in the Java DL calculus is based on symbolic program execution and simple program transformations and so is close to a programmer's understanding of Java. It can be seen as a modal logic with a modality $\langle \mathtt{p} \rangle$ for every program `p`, where $\langle \mathtt{p} \rangle$ refers to the final state (if `p` terminates normally) that is reached after executing `p`.

The *program formula* $\langle \mathtt{p} \rangle\, \phi$ expresses that the program `p` terminates in a state in which ϕ holds without throwing an exception. A formula $\phi \rightarrow \langle \mathtt{p} \rangle\, \psi$ is valid if for every state $\mathcal{S}$ satisfying precondition ϕ a run of the program `p` starting in $\mathcal{S}$ terminates normally, and in the terminating state the postcondition ψ holds.

The programs in Java DL formulas are basically executable Java code. Each rule of the Java DL calculus specifies how to execute symbolically one particular statement, possibly with additional restrictions. When a loop or a recursive method call is encountered, it is in general necessary to perform induction over a suitable data structure. In this paper we show how induction can be avoided in the case of parallelizable loops.

2.2 State Updates

In Java (as in other object-oriented programming languages), different object type variables may refer to the same object. This phenomenon, called aliasing, causes difficulties for handling of assignments in a calculus for Java DL. For example, whether or not the formula `o1.f` $\doteq$ `1` holds after (symbolic) execution of the assignment `o2.f = 2;`, depends on whether `o1` and `o2` refer to the same object. Therefore, Java assignments cannot be symbolically executed by syntactic substitution. In the Java DL calculus a different solution is used, based on the notion of (state) *updates*.

Definition 1. *Atomic updates* are of the form `loc` $:=$ `val`, where `val` is a logical term without side effects and `loc` is either (i) a program variable `v`, or (ii) a field access `o.f`, or (iii) an array access `a[i]`. Updates may appear in front of any formula, where they are surrounded by curly brackets for easy parsing. The semantics of $\{\texttt{loc} := \texttt{val}\}\phi$ is the same as that of $\langle\texttt{loc=val;}\rangle\,\phi$.

Definition 2. General *updates* are defined inductively based on atomic updates. If $\mathcal{U}$ and $\mathcal{U}'$ are updates then so are: (i) $\mathcal{U},\mathcal{U}'$ (*parallel composition*), (ii) $\mathcal{U};\mathcal{U}'$ (*sequential composition*), (iii) $\backslash\texttt{if}\ (b)\ \{\mathcal{U}\}$, where b is a quantifier-free formula (*conditional execution*), (iv) $\backslash\texttt{for}\ T\ s; \mathcal{U}(s)$, where s is a variable over a well-ordered type T and $\mathcal{U}(s)$ is an update with occurrences of s (*quantification*).

The semantics of sequential and conditional updates is obvious; the meaning of a parallel update is the simultaneous application of all its constituent updates except when two left hand sides refer to the same location: in this case the syntactically later update wins. This models natural program execution flow. The semantics of $\backslash\texttt{for}\ T\ s;\ \mathcal{U}(s)$ is the parallel execution of all updates in $\bigcup_{x\in T}\{s := x; \mathcal{U}(s)\}$. As for parallel updates, a last-win clash-semantics is in place: the maximal update with respect to the well-order on T and the syntactic order within each $\mathcal{U}(s)$ wins.

The restriction that right-hand sides of updates must be side effect-free is not essential: by introducing fresh local variables and symbolic execution of complex expressions the Java DL calculus rules normalize arbitrary assignments so that they meet the restrictions of updates. A full formal treatment of updates is in [Rö6].

3 Outline of the Approach

Let us look at the following example:

```
for (int i = 1; i < a.length; i++)
     if (c != 0) a[i] = b[i+1];
     else a[i] = b[i-1];
```

In a first step the loop initialization expression is transformed out of the loop and symbolically executed. The reason is that the initialization expression might be complex and have side effects. This results in a state $\mathcal{S} = \{\mathtt{i} := 1\}$. The remaining loop now has the form: `for (; i < a.length; i++)...`

We proceed to symbolically execute the loop body, the step expression and the guard for a generic value of `i`. In order to do this correctly, we must eliminate from the current state all locations that can potentially be modified in the body, step, or guard. In Section 4 we describe an algorithm that approximates such a set of locations rather precisely. Applied to the present example we obtain `i` and `a[i]` as modifiable locations. Consequently, generic execution of the loop body, step, and guard starts in the empty state. Note that the set of modifiable locations does not include, for example, `c`. This is important, because if $\mathcal{S}$ contains, say, $\mathtt{c} := 1$, we would start the execution in the state $\{\mathtt{c} := 1\}$ and the resulting state would be much simplified.

In our example, symbolic execution of one loop iteration starting in the empty state gives $\mathcal{S}' = \{\mathtt{i} := \mathtt{i} + 1,\ \backslash\mathtt{if}\ \ (c \neq 0)\ \{\mathtt{a[i]} := \mathtt{b[i+1]}\},\ \backslash\mathtt{if}\ \ (c \doteq 0)\ \{\mathtt{a[i]} := \mathtt{b[i-1]}\}\}$, where the step and guard expressions were executed as well.

The next step is to check whether the state update $\mathcal{S}'$ resulting from the execution of the generic iteration contains dependencies that make it impossible to represent the effect of the loop as a quantified update. For $\mathcal{S}'$ this is the case if and only if `c` is 0 and `a` and `b` are the same array. In this case, the body amounts to the statement `a[i] = a[i-1]` which contains a data dependence that cannot be parallelized. All other dependencies can be captured by parallel execution of updates with last-win clash-semantics. The details of the dependence analysis are explained in Section 6. In the example it results in a logical constraint $\mathcal{C}$ that, among other things, contains the disjunction of $\mathtt{c} \neq 0$ and $\mathtt{a} \neq \mathtt{b}$. A further logical constraint $\mathcal{D}$ strengthening $\mathcal{C}$ is computed which, in addition, ensures that the loop terminates normally. In the example, normal termination is ensured by `a` and `b` not being **null** and `b` having enough elements, that is, $\mathtt{b.length} > \mathtt{a.length}$.

At this point the proof is split into two cases using cut formula $\mathcal{D}$. Under the assumption $\mathcal{D}$ the loop can be transformed into a quantified update. If $\mathcal{D}$ is not provable, then the loop must be also tackled with a conventional induction rule, but one may use the additional assumption $\neg\mathcal{D}$, which may well simplify the proof.

For the sake of illustration assume now $\mathcal{S}$ and $\mathcal{S}'$ both contain $\{\mathtt{c} := 1\}$ and the termination constraint in $\mathcal{D}$ holds. In this case, we can additionally simplify $\mathcal{S}'$ to $\{\mathtt{c} := 1,\ \mathtt{i} := \mathtt{i} + 1,\ \mathtt{a[i]} := \mathtt{b[i+1]}\}$.

In the final step we synthesize from (i) the initial state $\mathcal{S}$, (ii) the effect of a generic execution of an iteration $\mathcal{S}'$ and (iii) the guard, a state update, where the loop variable `i` is universally quantified. The details are explained in Section 7. The result for the example is:

$$\backslash\mathtt{for}\ \textbf{int}\ I;\ \{\mathtt{i} := I; \backslash\mathtt{if}\ \ (\mathtt{i} \geq 1 \wedge \mathtt{i} < \mathtt{a.length})\ \{\mathtt{c} := 1,\ \mathtt{i} := \mathtt{i}{+}1,\ \mathtt{a[i]} := \mathtt{b[i+1]}\}\}$$

The `for`-expression is a universal first order quantifier whose scope is an update that contains occurrences of the variable `i` (see Def. 2 and [RÖ6]). Subexpressions are first order terms that are simplified eagerly while symbolic execution proceeds. First order quantifier elimination rules based on skolemization and instantiation are applicable, for example, for any positive value j such that $j <$ `a.length` we obtain immediately the update `a[`j`]` $:=$ `b[`j`+1]` by instantiation. Proof search is performed by the usual first order strategies without user interaction.

4 Computing state modifications

In this section we describe how we compute the state modifications performed by a generic loop iteration. As a preliminary step we move the initialization out of the loop and execute it symbolically, because the initialization expression may contain side-effects. We are left with a loop consisting of a guard, a step expression and a body:

```
for (; guard; step) body                                   (1)
```

We want to compute the state modifications performed by a generic iteration of the loop. A single loop iteration consists of executing the body, evaluating the step expression, and testing the guard expression. This behavior is captured in the following compound statement where `dummy` is needed, because Java expressions are not statements.

```
body; step; boolean dummy = guard;                         (2)
```

We proceed to symbolically execute the compound statement (2) for a generic value of the loop variable. This is quite similar to computing the strongest post condition of a given program. Platzer [Pla04] has worked out the details of how to compute the strongest post condition in the specific Java program logic that we use and our methods are based on the same principles.

Let `p` be the code in (2). The main idea is to try to prove validity of the program formula $\mathcal{S}\langle\mathtt{p}\rangle\, \mathit{fin}$, where *fin* is an arbitrary but unspecified non-rigid predicate that signifies when to stop symbolic execution. Symbolic execution of `p` starting in state $\mathcal{S}$ eventually yields a proof tree whose open leaves are of the form $\Gamma \to \mathcal{U}\, \mathit{fin}$ for some update expression $\mathcal{U}$. The predicate *fin* cannot be shown to be true or false in the program logic. Therefore, after all instructions in `p` have been executed, symbolic execution is stuck. At this stage we extract two vectors $\vec{\Gamma}$ and $\vec{\mathcal{U}}$ consisting of corresponding Γ and $\mathcal{U}$ from all open leaf nodes. Different leaves correspond to different computation branches in the loop body.

Example 1. Consider the following statement `p`:

```
if (i > 2) a[i] = 0 else a[i] = 1; i = i + 1;
```

After the attempt to prove $\langle\mathtt{p}\rangle\, \mathit{fin}$ becomes stuck there are two open leaves:

$$\begin{array}{lcl} V \,\wedge\, i > 2 & \to & \{\mathtt{a[i]} := 0,\, \mathtt{i} := \mathtt{i}+1\}\, \mathit{fin} \\ V \,\wedge\, i \not> 2 & \to & \{\mathtt{a[i]} := 1,\, \mathtt{i} := \mathtt{i}+1\}\, \mathit{fin} \end{array}$$

where V stands for $\neg(\mathtt{a} = \mathbf{null}) \wedge i \geq 0 \wedge \mathtt{i} < \mathtt{a.length}$. From these we extract the following vectors:

$$\begin{array}{lcl} \vec{\Gamma} & \equiv & \langle V \wedge i > 2, V \wedge i \not> 2\rangle \\ \vec{\mathcal{U}} & \equiv & \langle\{\mathtt{a[i]} := 0,\, \mathtt{i} := \mathtt{i}+1\},\, \{\mathtt{a[i]} := 1,\, \mathtt{i} := \mathtt{i}+1\}\rangle \end{array}$$

□

If the loop iteration throws an exception, abruptly terminates the loop, or when the automatic strategies are not strong enough to execute all instructions in `p` to completion, some open leaf will contain unhandled instructions and be of a form different from $\Gamma \rightarrow \mathcal{U}\,\mathit{fin}$. We call these *failed leaves* in contrast to leaves of the form $\Gamma \rightarrow \mathcal{U}\,\mathit{fin}$ that are called *successful*.

If a failed leaf can be reached from the initial state, our method cannot handle the loop. We must, therefore, make sure that our method is only applied to loops for which we have proven that no failed leaf can be reached. In order to do this we create a vector $\vec{\mathcal{F}}$ consisting of the Γ extracted from all failed leaves and let the negation of $\vec{\mathcal{F}}$ become a condition that needs to be proven when applying our method.

Example 2. In Example 1 only the successful leaves are shown. When the proof attempt becomes stuck, there are in addition failed leaves of following form:

$$\begin{array}{lcl} \mathtt{a} \doteq \mathbf{null} & \rightarrow & \ldots\ \mathit{fin} \\ \mathtt{a} \not\doteq \mathbf{null} \wedge \mathtt{i} < 0 & \rightarrow & \ldots\ \mathit{fin} \\ \mathtt{a} \not\doteq \mathbf{null} \wedge \mathtt{i} \not< \mathtt{a.length} & \rightarrow & \ldots\ \mathit{fin} \end{array}$$

From these we extract the following vector:

$$\vec{\mathcal{F}} \quad \equiv \quad \langle \mathtt{a} \doteq \mathbf{null},\, \mathtt{a} \not\doteq \mathbf{null} \wedge \mathtt{i} < 0,\, \mathtt{a} \not\doteq \mathbf{null} \wedge \mathtt{i} \not< \mathtt{a.length}\rangle$$

□

Note that symbolic execution discards any code that cannot be reached. As a consequence, an exception that occurs at a code location that cannot be reached from the initial state will not occur in the leaves of the proof tree. This means that our method is not restricted to code that cannot throw any exception, which would be very restrictive.

So far we said nothing about the state in which we start a generic loop iteration. Choosing a suitable state requires some care, as the following example shows.

Example 3. Consider the following code:

```
c = 1;
i = 0;
for (; i < a.length; i++) {
    if (c != 0) a[i] = 0;
    b[i] = 0; }
```

At the beginning of the loop we are in state $\mathcal{S} = \{\texttt{c} := 1,\ \texttt{i} := 0\}$. It is tempting, but wrong, to start the generic loop iteration in this state. The reason is that `i` has a specific value, so one iteration would yield $\{\texttt{a[0]} := 0,\ \texttt{b[0]} := 0,\ \texttt{i} := 1\}$, which is the result after the *first* iteration, not a generic one. The problem is that $\mathcal{S}$ contains information that is not invariant during the loop. Starting the loop iteration in the empty state is sound, but suboptimal. In the example, we get $\{\backslash\texttt{if}\ \ (\texttt{c} \neq 0)\ \{\texttt{a[i]} := 0\},\ \texttt{b[i]} := 0,\ \texttt{i} := \texttt{i} + 1\}$, which is unnecessarily imprecise, since we know that `c` is equal to 1 during the entire execution of the loop. □

We want to use as much information as possible from the state $\mathcal{S}_{\text{init}}$ at the beginning of the loop and only remove those parts that are not invariant during all iterations of the loop. Executing the loop in the largest possible state corresponds to performing dead code elimination. When we reach a loop of the form (1) in state $\mathcal{S}_{\text{init}}$ we proceed as follows:

1. Execute **boolean** `dummy = guard;` in state $\mathcal{S}_{\text{init}}$ and obtain $\mathcal{S}$. We need to evaluate the guard since it may have side effects. Evaluation of the guard might cause the proof to branch, in which case we apply the following steps to *each* branch. If our method cannot be applied to at least one of the branches we backtrack to state $\mathcal{S}_{\text{init}}$ and use the standard rules to prove the loop.
2. Compute the vectors $\vec{\Gamma}$, $\vec{\mathcal{U}}$ and $\vec{\mathcal{F}}$ from (2) starting in state $\mathcal{S}$.
3. Obtain $\mathcal{S}'$ by removing from $\mathcal{S}$ all those locations that are modified in a successful leaf, more formally: $\mathcal{S}' = \{(\ell := e) \in \mathcal{S} \mid \ell \notin \mathit{mod}(\vec{\mathcal{U}})\}$, where $\mathit{mod}(\vec{\mathcal{U}})$ is the set of locations whose value in $\vec{\mathcal{U}}$ differs from its value in $\mathcal{S}$.
4. If $\mathcal{S} = \mathcal{S}'$ then stop; otherwise let $\mathcal{S}$ become $\mathcal{S}'$ and goto step 2.

The algorithm terminates since the number of locations that can be removed from the initial state is bound both by the textual size of the loop[2]and, in case the state does not contain any quantified update, the size of the state itself. The final state of this algorithm is a greatest fixpoint containing as much information as possible from the initial state $\mathcal{S}$. Let us call this final state $\mathcal{S}_{\text{iter}}$.

Example 4. Example 3 yields the following sequence of states:

Round	Start state	State modifications	New state	Remark
1	$\{\texttt{c} := 1,\ \texttt{i} := 0\}$	$\{\texttt{a[0]} := 0,\ \texttt{b[0]} := 0,\ \texttt{i} := 1\}$	$\{\texttt{c} := 1\}$	
2	$\{\texttt{c} := 1\}$	$\{\texttt{a[i]} := 0,\ \texttt{b[i]} := 0,\ \texttt{i} := \texttt{i+1}\}$	$\{\texttt{c} := 1\}$	Fixpoint

□

Computing the set $\mathit{mod}(\vec{\mathcal{U}})$ can be difficult. Assume $\mathcal{S}$ contains $\texttt{a[c]} := 0$ and $\vec{\mathcal{U}}$ contains $\texttt{a[i]} := 1$. If `i` and `c` can have the same value then `a[c]` should

[2]Including the size of any method called by the loop.

be removed from $\mathcal{S}$, otherwise it is safe to keep it. In general it is undecidable whether two variables can assume the same value. One can use a simplified version of the dependence analysis described in Section 6 (modified to yield always a boolean answer) to obtain an approximation of location collision. The dependence analysis always terminates so this does not change the overall termination behavior.

A similar situation occurs when $\mathcal{S}$ contains $\texttt{a.f} := 0$ and $\vec{\mathcal{U}}$ contains $\texttt{b.f} := 1$. If `a` and `b` are references to the same object then `a.f` must be removed from the new state. Here we make a safe approximation and remove `a.f` unless we can show that `a` and `b` refer to different objects.

5 Loop Variable and Loop Range

For the dependence analysis and also later for creating the quantified state update we need to identify the loop variable and the loop range. In addition, we need to know the value that the loop variable has in each iteration of the loop, that is, the function from the iteration number to the value of the loop variable in that iteration. This is a hard problem in general, but whenever the loop variable is incremented or decremented with a constant value in each iteration, it is easy to construct this function. At present we impose this as a restriction: the update of the loop variable must have the form `l := l op e`, where `l` is the loop variable and `e` is invariant during loop execution. It would be possible to let the user provide this function at the price of making the method less automatic.

To identify the loop variable we compute a set of candidate pairs (`l`, `e`) where `l` is a location that is assigned the expression `e`, satisfying the above restriction, in all successful leaf nodes of the generic iteration. Formally, this set is defined as $\{(\texttt{l}, \texttt{e}) \mid \bigwedge_{\mathcal{U}\in\vec{\mathcal{U}}}\{\texttt{l} := \texttt{e}\} \in \mathcal{U}\}$. The loop variable is supposed to have an effect on the loop range; therefore, we remove all those locations from the candidate set that do not occur in the guard. If the resulting set consists of more than one location, we arbitrarily choose a syntactically minimal one (for example, `i` is regarded as smaller than `a[c]`).

The remaining candidates should be eliminated because they will all cause data flow-dependence. A candidate is eliminated by transforming its expression into an expression which is not dependent on the candidate location. For example, the candidate `l`, introduced by the assignment `l = l + c;`, can be eliminated by transforming the assignment into `l = init + I * c;`, where `init` is the initial value of `l` and `I` the iteration number.

Example 5. Consider the code in Example 1 which gives the following vector $\vec{\mathcal{U}}$ of updates occurring in successful leaves:

$$\vec{\mathcal{U}} \quad \equiv \quad \langle\{\texttt{a[i]} := 0, \texttt{i} := \texttt{i} + 1\}, \{\texttt{a[i]} := 1, \texttt{i} := \texttt{i} + 1\}\rangle$$

We identify the location `i` as the loop variable, assuming that `i` occurs in the guard. □

To determine the loop range we begin by computing the specification of the guard in a similar way as we computed the state modifications of a generic iteration in the previous section. We attempt to prove $\langle$**boolean** `dummy = guard;`$\rangle$ *fin*. From the open leaves of the form $\Gamma \rightarrow \{\texttt{dummy} := \texttt{e}, \ldots\}\, \mathit{fin}$, we create the formula *GS* which characterizes when the guard is true. Formally, *GS* is defined as $\bigvee_{\Gamma \in \vec{\Gamma}}(\Gamma \ \wedge \ \texttt{e} \doteq \mathbf{true})$. The formula *GF* characterizes when the guard is not successfully evaluated. We let *GF* be the disjunction of all Γ from the open leaves that are not of the form above.

Example 6. Consider the following guard `g` = `i < a.length`. When the attempt to prove $\langle$**boolean** `dummy = g;`$\rangle$ *fin* becomes stuck there are two successful leaves:

$$\begin{array}{lcl} \texttt{a} \not\doteq \mathbf{null} \ \wedge \ \texttt{i} < \texttt{a.length} & \rightarrow & \{\texttt{dummy} := \mathbf{true}\}\, \mathit{fin} \\ \texttt{a} \not\doteq \mathbf{null} \ \wedge \ \texttt{i} \not< \texttt{a.length} & \rightarrow & \{\texttt{dummy} := \mathbf{false}\}\, \mathit{fin} \end{array}$$

From these we extract the following formula *GS* (before simplification):

$$\begin{array}{l} (\texttt{a} \not\doteq \mathbf{null} \ \wedge \ \texttt{i} < \texttt{a.length} \ \wedge \ \mathbf{true} \doteq \mathbf{true}) \vee \\ (\texttt{a} \not\doteq \mathbf{null} \ \wedge \ \texttt{i} \not< \texttt{a.length} \ \wedge \ \mathbf{false} \doteq \mathbf{true}) \end{array}$$

When the attempt gets stuck, there is also the failed leaf $\texttt{a} \doteq \mathbf{null} \ \rightarrow \ \ldots \ \mathit{fin}$. From it we extract the following formula $GF \ \equiv \ \texttt{a} \doteq \mathbf{null}$. □

After having computed the specification of the guard and identified the loop variable we determine the initial value *start* of the loop variable from the initial state $\mathcal{S}_{\text{init}}$. If an initial value cannot be found we let it be unknown. We try to determine the final value *end* of the loop variable from the successful leaves of the guard specification. Currently, we restrict this to guards of the form `l` op `e`. When we cannot determine the final value, we let it be unknown. We already have determined the *step* value during identification of the loop variable.

The formula *GR* characterizes when the value of the loop variable `i` is within the loop range. It is defined as $GS \ \wedge \ \texttt{\textbackslash exists}\ \mathit{int}\ k; \texttt{i} \doteq \mathit{start} + k * \mathit{step}$.

It is important that the loop terminates, otherwise, our method is unsound. We create a termination constraint *GT* that needs to be proven when applying our method. The termination constraint says that the step is not equal to zero and there exists a value for the loop variable, a multiple of the step from the initial value, for which the guard formula is false. The constraint *GT* is defined as:

$$\mathit{step} \not\doteq 0 \ \wedge \ \texttt{\textbackslash exists}\ \mathit{int}\ I; \neg\{\texttt{i} := \mathit{start} + I * \mathit{step}\}GS$$

6 Dependence Analysis

Transforming a loop into a quantified state update is only possible when the iterations of the loop are independent of each other. Two loop iterations are independent of each other if the execution of one iteration does not affect the execution of the other. According to this definition, the loop variable clearly causes independence, because each iteration both reads its current value and updates it. We will, however, handle the loop variable by quantification. Therefore, it is removed from the update before the dependence analysis is begun. The problem of loop dependencies was intensely studied in loop vectorization and parallelization for program optimization on parallel architectures. Some of our concepts are based on results in this field [BCKT79, Wol89].

6.1 Classification of Dependencies

In our setting we encounter three different kinds of dependence; *data flow-dependence*, *data anti-dependence*, and *data output-dependence*.

Example 7. It is tempting to assume that it is sufficient for independence of loop iterations that the final state after executing a loop is independent of the order of execution, but the following example shows this to be wrong:

```
for (int i = 0, sum = 0; i < a.length; i++) sum += a[i];
```

The loop computes the sum of all elements in `a` which is independent of the order of execution, however, running all iterations in parallel gives the wrong result, because reading and writing of `sum` collide. □

Definition 3. Let $\mathcal{S}_J$ be the final state after executing a generic loop iteration over variable `i` during which it has value J and let $<$ be the order on the type of `i`.

There is a *data input-dependence* between iterations $K \neq L$ iff $\mathcal{S}_K$ writes to a location (ie, appears on the left-hand side of an update) that is read (appears on the right hand side or in a guard of an update) in $\mathcal{S}_L$. We speak of *data flow-dependence* when $K < L$ and of *data anti-dependence*, when $K > L$. There is *data output-dependence* between iterations $K \neq L$ iff $\mathcal{S}_K$ writes to a location that is overwritten in $\mathcal{S}_L$.

Example 8. When executing the second iteration of the following loop, the location `a[1]` which was modified by the first iteration is read, showing that there is a data flow-dependence:

```
for (int i = 1; i < a.length; i++) a[i] = a[i - 1];
```

The following loop exhibits data output-dependence:

```
for (int i = 1; i < a.length; i++) last = a[i];
```

Each iteration assigns a new value to `last`. When the loop terminates, `last` has the value assigned to it by the last iteration. □

Loops with data flow-dependencies cannot be parallelized, because each iteration must wait for a preceding one to finish before it can perform its computation.

In the presence of data anti-dependence swapping two iterations is unsound, but parallel execution is possible provided that the generic iteration acts on the original state before loop execution begins. In our translation of loops into quantified state updates in Section 7 below, this is ensured by simultaneous execution of all updates. Thus, we can handle loops that exhibit data anti-dependence. The final state of such loops depends on the order of execution, so independence of the order of executions is not only insufficient (Example 7) but even unnecessary for parallelization.

Even loops with data output-dependence can be parallelized by assigning an ordinal to each iteration. An iteration that wants to write to a location first ensures that no iteration with higher ordinal has already written to it. This requires a total order on the iterations. As we know the step expression of the loop variable, this order can easily be constructed. The order is used in the quantified state update together with a last-win clash-semantics to obtain the desired behavior.

6.2 Comparison to Traditional Dependence Analysis

Our dependence analysis is different from most existing analyses for loop parallelization in compilers [BCKT79, Wol89]. The major difference is that these analyses must not be expensive in terms of computation time, because the user waits for the compiler to finish. Traditionally, precision is traded off for cost. Here we use dependence information to avoid using induction which comes with an extremely high cost, because it typically requires user interaction. In consequence, we strive to make the dependence analysis as precise as possible as long as it is still fully automatic. In particular, our analysis can afford to try several algorithms that work well for different classes of loops.

A second difference to traditional dependence analysis is that we do not require a definite answer. When used during compilation to a parallel architecture, a dependence analysis must give a Boolean answer as to whether a given loop is parallelizable or not. In our setting it is useful to know that a loop is parallelizable relative to satisfaction of a symbolic constraint. Then we can let a theorem prover validate or refute this constraint, which typically is a much easier problem than proving the original loop.

6.3 Implementation

Our dependence analysis consists of two parts. The first part analyzes the loop and symbolically computes a *constraint* that characterizes when the loop is free of dependencies. The advantage of the constraint-based approach is that we can avoid to deal with a number of very hard problems such as aliasing: for example, locations `a[i]` and `b[i]` are the same iff `a` and `b` are references to the same array, which can be difficult to determine. Our analysis side-steps the aliasing problem simply by generating a constraint saying that *if* `a` is not the same array as `b` *then* there is no dependence. The second part of the dependence analysis is a tailor-made theorem prover that simplifies the integer equations occurring in the resulting constraints as much as possible.

The computation of the dependence constraints uses the vectors $\vec{\Gamma}$ and $\vec{\mathcal{U}}$ that represent successful leaves in the symbolic execution of the loop body and were obtained as the result of a generic loop iteration in Section 4. Let Γ_k and $\mathcal{U}_k$ be the precondition, respectively, the resulting update in the k-th leaf. If the preconditions of two leaves are true for different values in the loop range we need to ensure that the updates of the leaves are independent of each other (Def. 3). Formally, if there exist two distinct values K and L in the loop range and (possibly identical) leaves r and s, for which $\{\mathtt{i} := K\}\Gamma_r$ and $\{\mathtt{i} := L\}\Gamma_s$ are true, then we need to ensure independence of $\mathcal{U}_r$ and $\mathcal{U}_s$. We run our dependence analysis on $\mathcal{U}_r$ and $\mathcal{U}_s$ to compute the dependence constraint $\mathcal{C}_{r,s}$.

We do this for all pairs of leaves and define the dependence constraint for the entire loop as follows where GR is the loop range predicate:

$$\mathcal{C} \equiv \bigwedge_{r,s} \left(\left(\exists K, L. \left(\begin{array}{c} K \doteq\!\!\!\!\!/\; L \wedge \{\mathtt{i} := K\}GR \wedge \{\mathtt{i} := L\}GR \wedge \\ \{\mathtt{i} := K\}\Gamma_r \wedge \{\mathtt{i} := L\}\Gamma_s \end{array} \right) \right) \rightarrow \mathcal{C}_{r,s} \right)$$

Example 9. Consider the following loop that reverses the elements of the array `a`:

```
int half = a.length / 2 - 1;
for (int i = 0; i <= half; i++) {
  int tmp = a[i];
  a[i] = a[a.length - 1 - i];
  a[a.length - 1 - i] = tmp; }
```

When running the dependence analysis we get the following constraint:

$$\mathcal{C}_{0,0} \quad \equiv \quad \mathtt{a.length} < 2 \;\vee\; \mathtt{half} * 2 < \mathtt{a.length}$$

For this loop, the state $\mathcal{S}_{iter}$ contains `half` $:=$ `a.length / 2 - 1` and the constraint is, therefore, simplified to $\mathtt{a.length} < 2 \vee (\mathtt{a.length}/2)*2 < \mathtt{a.length}+2$. This is simplified to **true** which makes $\mathcal{C}$ true and means that the loop does not contain any dependencies that cannot be handled by our method. □

7 Constructing the State Update

If we can show that the iterations of a loop are independent of each other (that is, the constraint $\mathcal{C}$ defined in the previous section holds), we can capture all state modifications of the loop in a state update (Def. 2). Concretely, we use the following quantified update (T is the type of the loop variable `i`; GR, Γ_r, $\mathcal{U}_r$ were defined in Sections 4 and 5):

$$\mathcal{U}_{loop} \equiv \backslash\mathtt{for}\ T\ I;\ \{\mathtt{i} := I;\ \backslash\mathtt{if}\ (GR)\ \{\bigcup_r \backslash\mathtt{if}\ (\Gamma_r)\ \{\mathcal{U}_r\}\}\} \tag{3}$$

The conditional update inside (3) corresponds to one loop iteration, where `i` has the value I. In each state only one Γ can be true so we do not need to ensure any particular order of the updates $\vec{\mathcal{U}}$.

The guard GR ensures that `i` is within the loop range. We must take care when using last-win clash-semantics to handle data output-dependence. When the step is positive, the iteration with the highest value of the loop variable should have priority over all other iterations. This is ensured by the standard well-order on the JAVA integer types.

A complication arises when the step is negative. Then we need to reverse the order so that the iteration with the lowest value of the loop variable has priority. Since each type has a fixed order we need to change the state update instead: it is sufficient to replace in (3) the update $\mathtt{i} := I$ with $\mathtt{i} := -I$.

8 Using the Analysis in a Correctness Proof

When we encounter a loop during symbolic execution we analyze it for parallelizability as described above and compute the dependence constraint. We replace the loop by (3) if no failed leaves for the iteration statement or the guard expression can be reached (see Section 4), the loop terminates (formula GT, see Section 5), and the dependence constraint $\mathcal{C}$ in Section 6.3 is valid. Taken together, this yields:

$$\mathcal{D} \equiv (\bigwedge_i \neg(\exists I.\mathcal{F}_i[I])) \wedge \neg(\exists I.GF[I]) \wedge GT \wedge \mathcal{C}$$

If $\mathcal{D}$ does not hold, we fall back to the standard rules to verify the loop (usually induction). In many cases it is not trivial to immediately validate or refute $\mathcal{D}$. Then we perform a cut on $\mathcal{D}$ in the proof and replace the loop by the quantified state update $\mathcal{U}_{loop}$ (3) in the proof branch where $\mathcal{D}$ is assumed to hold. The general outline of a proof using a cut on $\mathcal{D}$ is as follows:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\text{If not } \Gamma \ \Rightarrow \ \mathcal{D},\\ \text{use standard induction}\end{array}}{\Gamma \ \Rightarrow \ \mathcal{U}\langle \textbf{for} \ \ldots \ ; \ \ldots\rangle \, \phi, \mathcal{D}} \qquad \dfrac{\Gamma, \ \mathcal{D} \ \Rightarrow \ \mathcal{U}\mathcal{U}_{loop}\langle \ldots \rangle \, \phi}{\Gamma, \ \mathcal{D} \ \Rightarrow \ \mathcal{U}\langle \textbf{for} \ \ldots \ ; \ \ldots\rangle \, \phi}}{\Gamma \ \Rightarrow \ \mathcal{U}\langle \textbf{for} \ \ldots \ ; \ \ldots\rangle \, \phi} \, cut}{\vdots}$$

If we can validate or refute $\mathcal{D}$ we can close one of the two branches. Typically, this involves to show that there is no aliasing between the variables occurring in the dependence constraint. Even when it is not possible to prove or to refute $\mathcal{D}$ our analysis is useful, because $\mathcal{D}$ in succedent of the left branch can make it easier to close.

9 Evaluation

We evaluated our method with three representative JAVA CARD programs [Mos05]: DeMoney, SafeApplet and IButtonAPI that together consist of ca. 2200 lines of code (not counting comments). In these programs there exist 17 loops. Out of these, five can be handled (sometimes, a simple code transformation like `v += e` to `v = v0 + i * e` is required). Additionally, four loops can be handled if we allow object creation in the quantified updates (which is currently not realized). The remaining eight loops cannot be handled because they contain abrupt termination and irregular step functions. The results are summarized in the following table:

	DeMoney	SafeApplet	IButtonAPI	Total
LoC	1633	514	102	2249
Size (kB)	182	22	3	207
# loops	10	6	1	17
handled	4	0	1	5
with ext.	3	1	0	4
remaining	3	5	0	8

All loops in the row "handled" are detected automatically as parallelizable and are transformed into quantified updates.

The evaluation shows that a considerable number of loops in realistic legacy programs can be formally verified without resorting to interactive and, therefore, expensive techniques such as induction. Interestingly, the percentage of loops that can be handled differs drastically among the three programs. A closer inspection reveals that the reason is not that, for example, all the loops in SafeApplet are inherently not parallelizable. Some of them could be rewritten so that they become parallelizable. This suggests to develop programming guidelines (just as they exist for compilation on parallel architectures) that ensure parallelizability of loops.

10 Conclusion

We presented a method for formal verification of loops that works by transforming loops into automizable first order constructs (quantified updates) instead of interactive methods such as invariants or induction. The approach is restricted to loops that can be parallelized, but an analysis of representative programs from the Java Card domain shows that such loops occur frequently. For example, most initialization and array copy methods are based on parallelizable loops.

The method relies on the capability to represent state change information effecting from symbolic execution of imperative programs explicitly in the form of syntactic updates [Bec01, R0̈6]. With the help of updates the effect of a generic loop iteration is represented so that it can be analyzed for the presence of data dependencies. Ideas for the dependency analysis are taken from compiler optimization for parallel architectures, but the analysis is not merely static. Loops that are found to be parallelizable are transformed into first order quantified updates to be passed on to an automated theorem prover.

A main advantage of our method is its robustness in the presence of syntactic variability in the target programs. This is achieved by performing symbolic execution before doing the dependence analysis. The method is also fully automatic whenever it is applicable and gives useful results in the form of symbolic constraints even if it fails.

Future Work The analysis can be improved in various ways. One example is the function from iteration number to value of the loop variable (see Section 5). In addition, straightforward automatic program transformations that reduce the amount of dependencies (for example, `v += e;` into `v = vInit + i * e;`) could be derived by looking at the updates from a generic loop iteration. We also intend to develop general programming guidelines that ensure parallelizability of loops. Recent work on automatic termination analysis [CPR06] could be adapted to the present setting for proving the termination constraint in Section 5.

Critical dependencies exhibited during the analysis are likely to cause problems as well in a proof attempt based on invariants or induction, so one could try to use the obtained information on dependencies to guide the generalization of loop invariants.

At the moment we observe JAVA integer semantics only by checking for overflow. The integer model could be made more precise by computing all integer operators modulo the the size of the underlying integer type. This would require changes in the dependence analysis; the JAVA DL calculus covers full JAVA integer semantic already [BS04].

Finally, the discussion in this paper stops after a loop has been transformed into a quantified update. So far, our theorem prover has limited capabilities for automatic reasoning over first order quantified updates. Since quantified updates occur in many other scenarios it is worth to spend more effort on that front.

Acknowledgments Many thanks to Richard Bubel whose help with the implementation was invaluable! Thanks are also due to Philipp Rümmer for many inspiring discussions.

Bibliography

[ABB$^+$05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool: integrating object oriented design and formal verification. *Software and System Modeling*, 4(1):32–54, 2005.

[BBHI05] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, June 2005.

[BCKT79] Utpal Banerjee, Shyh-Ching Chen, David J. Kuck, and Ross A. Towle. Time and parallel processor bounds for Fortran-like loops. *IEEE Trans. Computers*, 28(9):660–670, 1979.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[Bre06] Cees-Bart Breunesse. *On JML: Topics in Tool-assisted Verification of Java Programs*. PhD thesis, Radboud University of Nijmegen, 2006.

[BS04] Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proc. , Intl. Conf. on Integrated Formal Methods*, volume 2999 of *LNCS*, pages 207–226. Springer, 2004.

[BSS05] Bernhard Beckert, Steffen Schlager, and Peter H. Schmitt. An improved rule for while loops in deductive program verification. In Kung-Kiu Lau, editor, *Proc. , Seventh Intl. Conf. on Formal Engineering Methods (ICFEM), Manchester, UK*, LNCS. Springer-Verlag, 2005.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM Press, to appear, 2006.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[HM05] Reiner Hähnle and Wojciech Mostowski. Verification of safety properties in the presence of transactions. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conf. Proc. of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.

[Hol02] Gerard J. Holzmann. Software analysis and model checking. In E. Brinksma and K. Guldstrand Larsen, editors, *Proc. Intl. Conf. on Computer-Aided Verification CAV, Copenhagen*, volume 2402 of *LNCS*, pages 1–16. Springer, July 2002.

[Mos05] Wojciech Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In Maura Cerioli, editor, *Proc. Fundamental Approaches to Software Engineering (FASE), Edinburgh*, volume 3442 of *LNCS*, pages 357–371. Springer, April 2005.

[OW05] Ola Olsson and Angela Wallenburg. Customised induction rules for proving correctness of imperative programs. In Bernhard Beckert and Bernhard Aichernig, editors, *Proc. , Software Engineering and Formal Methods (SEFM), Koblenz, Germany*, pages 180–189. IEEE Press, 2005.

[Pla04] André Platzer. Using a program verification calculus for constructing specifications from implementations. Master's thesis, Univ. Karlsruhe, Dept. of Computer Science, 2004.

[R0̈6] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. 2006. Submitted to this conference (IJCAR 2006).

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.