Thesis for the Degree of Doctor of Philosophy

Combining and Strengthening Program Analysis and Verification

Tobias Gedell

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, 2008

Combining and Strengthening Program Analysis and Verification
Tobias Gedell
ISBN 978-91-7385-158-9

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2839
ISSN 0346-718X

Technical Report 42D
Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31–772 10 00

Printed at Chalmers
Göteborg, Sweden 2008

Abstract

This thesis is about methods for establishing semantic properties of programs and how those methods can be strengthened. Finding (semi-)algorithms for deciding semantic properties is a non-trivial task and such algorithms will, by necessity, give approximate answers. This means that for any property of interest, there is a spectrum of algorithms computing answers to various degrees of precision, ranging from computationally cheap, low-precision algorithms to expensive, potentially non-terminating algorithms with very high precision. Finding approximations precise enough to be useful, and that at the same time make the algorithms cheap enough, is a real challenge.

In this thesis we consider *program analysis* and *program verification*, which are two approaches for establishing program properties with contrasting requirements regarding precision and cost. Common to these two approaches is the desire to move closer to the middle of the spectrum of algorithms. For algorithms formulated as program analyses, this means increasing their precision, and for algorithms formulated as program verifiers, it means making them terminate without user interaction for a larger set of programs.

The work presented in this thesis can be divided into three parts. The first part investigates the impact a number of features have on the precision of a type-based program analysis for Haskell. The results presented, make it easier for a designer of a type-based program analysis to choose what features to add to the analysis in order to get sufficiently high precision.

The second part concerns how program verification can be integrated with program analysis in order to make the verification cheaper and more automatic. We show how a program analysis can be embedded in the tactic language of a theorem prover to allow for a close integration of program analysis and verification. We also show, in particular, how a dependence analysis can be used in a theorem prover for Java to make the handling of loops more automatic.

The third part concerns how type-based program analysis can be strengthened by plugging in additional, externally computed information. We do this by presenting a modular method for parameterizing type-based program analyses over information about the analyzed program's execution. The parameterization is done in such a way that a modular proof of correctness is obtained, removing the need to prove correctness of each instantiation separately. We also show how our method of parameterization can be used to allow for flow-sensitive heap types by plugging in alias information.

Acknowledgments

First of all, I would like to thank my current and former supervisors, David Sands, Reiner Hähnle and Jörgen Gustavsson for all the help and guidance they have given me during my five years of PhD studies. I have learned a lot from them, and without them I could not have written this thesis.

Secondly, I would like to thank Daniel Hedin for the two years we have spent working together. It has been a very stimulating time with many long and fruitful discussions. I have particularly enjoyed all the time we have spent outside work.

In particular, I would like to thank John Hughes and Koen Claessen for their encouragement and help, and for being such great guys. John is the one who made me want to become a PhD student, for which I am very grateful.

It has been a real pleasure to work with my co-author Josef Svenningsson and he deserves a special thank. I would also like to thank the members of the KeY Group and the ProSec Group, all of whom I have worked together with during my time at Chalmers. All my other colleagues also deserve to be thanked for making Chalmers a nice place to work.

I am especially grateful to my girlfriend, Johanna, for being there for me, especially during the last half year when time has seemed way too scarce.

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

Tobias
London, August 2008

Contents

Chapter 1

Introduction

This thesis is about methods for establishing semantic properties of programs and how those methods can be strengthened. Examples of typical semantic properties are: *Will a variable be used more than once during execution? Will two variables point to the same object in the heap? Will secret information stored in one variable be leaked to another?*

Finding algorithms[1] for deciding this kind of properties is non-trivial and is the subject of a large body of research including this thesis. As a consequence of Rice's theorem [Ric53], which states that any formal system that is strong enough to capture a significant amount of the analyzed program's semantics will be undecidable, there are no terminating algorithms for deciding non-trivial properties. This means that if we want to have a terminating algorithm it will by necessity decide an approximation of the sought after property. Depending on what the algorithm is used for, there might also, besides termination, be constraints on the computational complexity of the algorithm. Imagine, for example, an algorithm that is used by a compiler to guide some optimizing transformations. Since users do not want to wait long for a compiler to finish, it is in this case more important that the algorithm terminates quickly than that it computes a very precise result. To fulfill these constraints the algorithm will need to approximate parts of the analyzed program's semantics.

Approximation can also be useful for non-terminating algorithms. Although they are not guaranteed to terminate for all programs, there is typically a set of programs for which they do terminate. By making the algorithms more approximate, this set of programs can be made larger.

This means that for any property of interest, there is a spectrum of algorithms, computing answers to various degrees of precision, ranging from computationally cheap, terminating algorithms with low precision to expensive, potentially non-terminating algorithms with very high precision. The spectrum of algorithms is illustrated in Figure 1.

[1] In this thesis, *algorithms* refers to both algorithms guaranteed to terminate and non-terminating semi-algorithms.

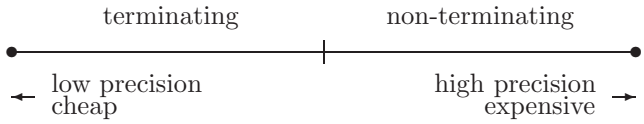


Figure 1: Spectrum of Algorithms

This is what makes this area of research so interesting. For a given property there is no such thing as the "best" approximation, it all comes down to tradeoffs between precision and cost, based on what the algorithm will be used for.

Two approaches for establishing properties with contrasting requirements regarding precision and cost are *program analysis* and *program verification*.

Program analysis [NNH99] is located on the left-hand side of the spectrum and consists of automatic and computationally cheap algorithms that terminate for all programs. Their prime usage is in situations where runtime is a major factor or where only limited resources are available. Typical examples include guiding optimizations in compilers and giving warnings about possible errors in programs. In these cases, missing an optimization or erroneously report an error can be tolerated. A missed optimization will only result in the compiled program running a little bit slower than best possible and a spurious warning can be inspected by the programmer and be ignored. None of which comes at a particular high cost.

Program verification by deduction based on logic and theorem proving [RV01, BHS07], is located on the right-hand side of the spectrum where precision is more important than complexity and termination. It is often used in situations where it is more costly to modify the program being verified than allowing the verification process more time and resources. A typical example of this is proving correctness of safety-critical programs. If a program is erroneously labeled as being incorrect it might be very costly to rewrite it and the rewritten program might still be erroneously labeled, forcing further rewrites. In this situations it can be cheaper to establish the property by proving it using a theorem prover and let a trained user interact with the proof creation. The chance of this succeeding is considerably higher given that the user probably has knowledge about the program which could not easily be discovered by a program analysis. Also, if the proof creation would fail, the user would have detailed information about what went wrong and would be able to more easily change the program accordingly.

When designing a program analysis for a particular semantic property the aim is typically to make it as precise as possible while keeping it tractable. In order to do so it has to be decided what parts of the program semantics should be abstracted and to what degree. For program analyses formulated as type systems, so called *type-based program analyses* [Pie02], there are a number of common features that are orthogonal to each other and each affect the level of

approximation and, thus, the precision of the analyses. Examples of such features are *polymorphism*, *subtyping*, *annotation of datatypes* and *whole program analysis*. It has, however, been unclear exactly how these features interact with each other and what impact they have on the precision. Simply adding all features will raise the precision but probably render the analysis too expensive for most uses. This is the subject of Paper I of this thesis in which we investigate what impact the above features have on the precision of a particular type-based program analysis, *usage analysis* [LGH+92]. The results of this paper make it easier for a designer of a type-based program analysis to decide what features are likely to have the largest impact on the precision and should be included.

Common to both program analysis and program verification is the desire to move closer to the middle of the spectrum of algorithms. For algorithms formulated as program analyses, this means increasing their precision, and for algorithms formulated as program verifiers, it means making them terminate without user interaction for a larger set of programs. One way to do that is by combining them with already existing program analyses. Consider for example a theorem prover used to prove some property of a program. It might well be the case that at some point, the theorem prover will require the user to assist with choosing how to proceed with the creation of the proof. If a cheap program analysis could figure out which choice to make, the need for user interaction, which is associated with a high cost, could be reduced. How this can be done is investigated in Paper II and Paper III of this thesis. In Paper II we show how a particular program analysis, *reaching definitions analysis* [NNH99], can be embedded in the tactic language of a theorem prover for JAVA. This allows for a close integration of theorem proving and program analysis and opens up for potential new areas of application. The embedding of the particular analysis is, however, only made as a proof of concept and the embedded analysis is not used by the theorem prover. In Paper III we go a step further and give a concrete example of how the theorem prover can interact with a program analysis by showing how the handling of loops can be made more automatic by the use of a *dependence analysis* [Wol89].

An important property of program analysis and program verification is *correctness*, i.e., that the computed results are sound approximations of the analyzed program's semantics. In both Paper II and Paper III the goal is to show how program analysis and program verification can be combined but correctness is not established. This is addressed in Paper IV and Paper V.

Paper IV and Paper V of this thesis concern how type-based program analyses can be strengthened by making available information computed by other program analyses. For example, knowing if a variable contains a null-pointer is a rather fundamental semantic property that is useful for a large number of analyses deciding more complex properties. There are two major different methods of strengthening a program analysis and making additional information available: by *integration* and by *parameterization*. Briefly, integration relies on extending the rules of the analysis to also compute the additional information and parameterization relies on extracting the information from the result of an external analysis.

Integration is problematic for a number of reasons: i) It obscures the original intention of the analysis since the rules are extended to compute additional information and, thus, grow more complex which make them harder to understand and maintain. This is especially true if multiple external analyses are integrated. ii) It makes usage of already available analyses difficult, since they have to be rephrased in such a way that their rules are compatible with the rules of the original analysis. iii) It is non-modular: changing the external analysis implies changing the entire combined analysis. An important consequence of this is that if the external analysis is changed then the entire proof of correctness of the combined analysis has to be redone.

Because of this, we believe parameterization to be the better method and address in Paper IV both the importance of keeping the original analysis and the external analysis separated and the need to establish correctness of the resulting analysis. We do this by presenting a modular method for parameterizing type-based program analyses over information about the analyzed program's execution. The parameterization is done in such a way that a modular proof of correctness is obtained.

In Paper V we use our method of parameterization to show how a flow-sensitive type-based program analysis can be made more precise by using *alias information* [CC77a, Coo85]. In particular, we show how alias information allows for floating heap types by introducing rules for *structural weakening* and *strong updates*.

Outline The outline of this chapter is as follows. Section 1 gives an introduction to program analysis. Section 1.1 shows how type systems can be used for program analysis and how correctness can be established, which sets the scene for Paper IV and Paper V. Section 1.2 introduces usage analysis, subtyping and polymorphism which are used in Paper I. In Section 2, program verification and the particular theorem prover used in Paper II and Paper III is introduced. Finally, Section 3 presents the papers included in this thesis in a bit more detail and Section 4 gives an overview of my personal contributions to the included papers.

1 Program Analysis

A central topic of this thesis is program analysis which consists of lightweight formal methods for deciding semantic properties of programs. Examples of such formal methods are *type systems* [Pie02], *abstract interpretation* [CC77b, CC79] and *model checking* [CE82, QS82]. In this thesis we mostly consider type-based program analysis and specifically the class of automatic and inexpensive analyses that are often used to guide program transformations and optimizations or establish relatively simple semantic properties. Two examples of program analyses in this class are *null-pointer analysis* [NNH99], that determines what variables in a program might contain null-pointers during execution, and *alias*

analysis [CC77a, Coo85, CWZ90], that determines for each pair of program locations if they could at runtime point to the same object.

An important characteristic of these two analyses, and program analyses in general, is that they terminate and give an answer for all programs. Unfortunately, virtually all interesting semantic properties are undecidable. A program analysis must, therefore, give approximate answers in order to fulfill the termination constraint. When analyses make such approximations, it is important that they are done in a *sound* way. This can be done in two ways, either by computing an *over-approximation* or computing an *under-approximation*.

Let S be the set of reachable states for a program being analyzed. An over-approximation overestimates the reachable states, computing a set S^+, such that $S^+ \supseteq S$. This is useful to establish that a property holds during the execution of the program; it is guaranteed to hold if it holds for all states in S^+. It might, however, result in *false negatives*, since there might be a state present in S^+ but not in S for which the property does not hold.

Under-approximations can be seen as the dual of over-approximations. They underestimate the reachable program states, computing a subset $S^- \subseteq S$. This is useful to establish that a property does not hold during the execution of the program, but might result in *false positives*.

Besides being sound, another important characteristic of the class of program analyses we consider is that they are computationally cheap, i.e. that they are not too resource or time consuming. Ideally, the complexity of a program analysis should be close to linear, at least when analyzing real world programs[2].

1.1 Type-Based Program Analysis

The program analyses considered in Paper I, Paper IV and Paper V of this thesis are formulated as type-based program analyses, which is the topic of this section. A type-based program analysis can be seen as an extension to a type system that in addition to the underlying type information also computes information about some sought after property.

In the book Types and Programming Languages [Pie02], Pierce defines type systems in the following way: "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute".

A type system classifies phrases by defining a typing relation that relates terms of the program language with types of a type language. Typically, if a term and a type is related by the typing relation, the type system guarantees that the term evaluates to a value of the specific type. A typical typing relation judgment is of the form $\Gamma \vdash e : \tau$ and expresses that the program fragment e has the type τ in the typing environment Γ.

The following example presents a type system for a language of expressions, showing how typing relations are typically defined inductively by a set of in-

[2]It will almost always be possible to craft contrived example programs for which a program analysis does not exhibit a linear complexity. This does, however, not affect its practical usefulness.

ference rules. This type system is later used to show how type-based program analyses are based on type systems and also to show how correctness of type systems can be established.

Example 1.1 (A Type System for Expressions) *Let the language of expressions consist of boolean literals, ranged over by b, integer literals, ranged over by i, variables, ranged over by x, addition, division and choice.*

$$\textbf{\textit{Expressions}} \quad e \ ::= \ b \mid i \mid x \mid e + e \mid e/e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$$

There are only two different types of values the expressions can evaluate to: integer values and boolean values. This is reflected by the type language which consists of only two types: the type for boolean values, `bool`*, and the type for integer values,* `int`*.*

$$\textbf{\textit{Types}} \quad \tau \ ::= \texttt{bool} \mid \texttt{int}$$

The typing relation is defined by the following inference rules where the typing environment Γ *is a map from variables to types.*

$$\frac{}{\Gamma \vdash b : \texttt{bool}} \qquad \frac{}{\Gamma \vdash i : \texttt{int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 / e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

The rules express that boolean and integer literals are well-typed with respect to `bool` *and* `int`*, respectively, the types of variables are given by the typing environment, and that the result of adding or dividing two integers is of type* `int`*. When branching over a boolean value we do not generally know what branch will be taken. This is reflected in the rule for choice which requires both branches to be of the same type which is also the type of the choice expression.* □

If $\Gamma \vdash e : \tau$ holds for an expression e, a type τ and a typing environment Γ, the type system guarantees that evaluation of e in an environment that conforms to Γ will not lead to any run-time type errors, such that trying to add a boolean value with an integer value. There are, however, run-time errors that are not caught by the type system them stem from division; if the rightmost operand evaluates to zero it causes a division by zero error.

One way to strengthen the type system to be able to catch these errors is by extending it, forming a type-based program analysis that besides computing the types of expressions also tracks whether the expressions might cause division by zero errors.

The following example presents an extension of the type system, forming a simple division by zero analysis. The judgments of the analysis are of the form

$\Gamma \vdash e : \tau, \pi$ where π ranges over the error annotations **ok**, expressing that the expression will not cause an error, and **error**, expressing that the expression might cause an error.

Example 1.2 (A Simple Division by Zero Analysis) *In the following rules we let $\sqcup$ be the least upper bound using the order $\textbf{\textit{ok}} < \textbf{\textit{error}}$ such that $\textbf{\textit{ok}} \sqcup \textbf{\textit{error}} = \textbf{\textit{error}}$.*

$$\frac{}{\Gamma \vdash b : \texttt{bool}, \textbf{\textit{ok}}} \qquad \frac{}{\Gamma \vdash i : \texttt{int}, \textbf{\textit{ok}}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau, \textbf{\textit{ok}}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int}, \pi_1 \quad \Gamma \vdash e_2 : \texttt{int}, \pi_2}{\Gamma \vdash e_1 + e_2 : \texttt{int}, \pi_1 \sqcup \pi_2} \qquad \frac{\Gamma \vdash e_1 : \texttt{int}, \pi_1 \quad \Gamma \vdash e_2 : \texttt{int}, \pi_2}{\Gamma \vdash e_1 / e_2 : \texttt{int}, \textbf{\textit{error}}}$$

$$\frac{\Gamma \vdash e : \texttt{bool}, \pi \quad \Gamma \vdash e_1 : \tau, \pi_1 \quad \Gamma \vdash e_2 : \tau, \pi_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau, \pi \sqcup \pi_1 \sqcup \pi_2}$$

The rules express that literal and variable expressions never cause any errors. Addition and choice expressions only cause errors if any of their subexpressions cause errors, which is reflected in the rules by the propagation of the error annotations. The rule for division approximates the semantic behavior very conservatively and assumes that all divisions cause errors.

□

This very naïve way of tracking errors is much too conservative to have any practical usage since every expression containing a division will be annotated as possibly resulting in an error. A real analysis would need to be strengthen with much more detailed information about, for example, what values the subexpressions evaluate to, in order to be able to identify divisions where the rightmost operand never evaluates to zero.

In Paper IV of this thesis we present a modular method for parameterizing type systems over such information about the analyzed program's execution. By using our method, we can replace the above rule for division by the following two rules, allowing for a higher precision.

$$\frac{\Gamma \vdash^{\mathbb{E}, R^{nz}} e_1 : \texttt{int}, \pi_1 \quad \Gamma \vdash^{\mathbb{E}, R^{nz}} e_2 : \texttt{int}, \pi_2 \quad R^{nz}_{\mathbb{E}}(e_2)}{\Gamma \vdash^{\mathbb{E}, R^{nz}} e_1 / e_2 : \texttt{int}, \pi_1 \sqcup \pi_2}$$

$$\frac{\Gamma \vdash^{\mathbb{E}, R^{nz}} e_1 : \texttt{int}, \pi_1 \quad \Gamma \vdash^{\mathbb{E}, R^{nz}} e_2 : \texttt{int}, \pi_2 \quad \neg R^{nz}_{\mathbb{E}}(e_2)}{\Gamma \vdash^{\mathbb{E}, R^{nz}} e_1 / e_2 : \texttt{int}, \textbf{error}}$$

The new rules are parameterized over an abstract environment $\mathbb{E}$, representing all concrete environments that the division might be evaluated in during execution, and a so-called *plugin* R^{nz} such that $R^{nz}_{\mathbb{E}}(e)$ holds if e is guaranteed to never evaluate to zero in any of the concrete environments abstracted by $\mathbb{E}$. If the side-condition $R^{nz}_{\mathbb{E}}(e_2)$ holds, it means that e_2 will never evaluate to zero and, thus, no division by zero error will occur. This is reflected in the first rule where the error annotation of the division is the least upper bound of the error

annotations of its subexpressions. If the side-condition does not hold, we cannot exclude the possibility that e_2 might evaluate to zero, and the type system falls back to the original behavior and uses the second type rule which assumes that the division will cause an error and annotates the expression with **error**.

Correctness Being correct is crucial for all program analyses and in contrast to, for example, abstract interpretations, type-based program analyses have no built-in correctness. Instead, once a type-based program analysis has been defined, correctness has to be established separately. This is a particular focus of Paper IV in which we consider how correctness of parameterized analyses can be established.

One way to establish correctness of type-based program analyses, and type systems in general, is to prove so-called *progress* and *preservation* [Pie02]. This is done with respect to a formal semantics of the program language, which is typically given as a small step semantics, and a well-formedness relation for values and variable environments.

We will here establish correctness of the above defined division by zero analysis and in order to do so, we give a formal semantics for the language of expressions.

Example 1.3 (A Semantics for Expressions) *Let v range over the values that consist of the integers, ranged over by i, and the booleans, ranged over by b.*

$$\textbf{\textit{Values}} \quad v \quad ::= i \mid b$$

The semantics of the expressions is given in terms of a small step semantics between configurations C with transitions of the form $\langle E, e\rangle \to C$ where C is either one of the terminal configurations $\bot$ and $\langle E, v\rangle$ indicating abnormal and normal termination respectively, or a non-terminal configuration $\langle E, e\rangle$ where E ranges over variable environments, i.e. maps from variables to values. The transition rules are defined as follows.

$$\frac{E(x) = v}{\langle E, x\rangle \to \langle E, v\rangle} \qquad \frac{v = v_1 + v_2}{\langle E, v_1 + v_2\rangle \to \langle E, v\rangle}$$

$$\frac{v_2 \neq 0 \qquad v = v_1/v_2}{\langle E, v_1/v_2\rangle \to \langle E, v\rangle} \qquad \frac{v_2 = 0}{\langle E, v_1/v_2\rangle \to \bot}$$

$$\frac{}{\langle E, \texttt{if}\ \mathit{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \to \langle E, e_1\rangle}$$

$$\frac{}{\langle E, \texttt{if}\ \mathit{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \to \langle E, e_2\rangle}$$

$$\frac{\langle E_1, e_1\rangle \to \langle E_2, e_2\rangle}{\langle E_1, R[e_1]\rangle \to \langle E_2, R[e_2]\rangle} \qquad \frac{\langle E, e\rangle \to \bot}{\langle E, R[e]\rangle \to \bot}$$

As is common for small step semantics we use reduction contexts, ranged over by R, to allow for inner reduction and propagation of errors.

$$R \quad ::= \quad \cdot \mid R/e \mid e/R \mid R + e \mid e + R \mid \texttt{if}\ R\ \texttt{then}\ e\ \texttt{else}\ e$$

□

Before giving the progress and preservation lemmas we need to define what it means for values, environments and configurations to conform to a type, i.e. we need to define *well-formedness relations*. We define our well-formedness relations as follows.

Example 1.4 (Well-formedness) *The judgments of the well-formedness relation for values, of the form* $\vdash v : \tau$*, express that the value* v *is well-formed with respect to the type* τ*.*

$$\frac{}{\vdash b : \mathtt{bool}} \qquad \frac{}{\vdash i : \mathtt{int}}$$

The rules for values express that boolean and integer values are well-typed with respect to their respective types.

The judgment of the well-formedness relation for variable environments, of the form $\vdash E : \Gamma$*, expresses that the environment* E *is well-formed with respect to the typing environment* Γ*.*

$$\frac{\forall x \in dom(\Gamma).\ \vdash E(x) : \Gamma(x)}{\vdash E : \Gamma}$$

The rule for environments expresses that the environment must map all variables in the typing environment to values which are well-formed with respect to the types given to them by the typing environment.

The judgments of the well-formedness relation for configurations, of the form $\vdash C : \tau, \Gamma, \pi$*, express that the configuration* C *is well-formed with respect to the type* τ*, the typing environment* Γ *and the error annotation* π*.*

$$\frac{}{\vdash \bot : \tau, \Gamma, \mathbf{error}} \qquad \frac{\vdash v : \tau}{\vdash v : \tau, \Gamma, \pi} \qquad \frac{\vdash E : \Gamma \quad \vdash e : \tau}{\vdash \langle E, e\rangle : \tau, \Gamma, \pi}$$

The rules state that the abnormal terminal configuration is only well-formed in the error annotation indicating error, a normal terminal configuration is well-formed if the value is well-formed, and a non-terminal configuration is well-formed if both the environment and expression are well-formed.

□

With this we are ready to give examples of progress and preservation lemmas.

Example 1.5 (Progress and Preservation) *The progress lemma states that evaluating a well-typed expression in a well-formed environment, results in a configuration. In the definition we let* e *range over non-literal expressions.*

$$\Gamma \vdash e : \tau, \pi \ \wedge\ \vdash E : \Gamma \implies \exists C.\ \langle E, e\rangle \to C$$

The preservation lemma states that the result of evaluating a well-typed expression in a well-formed environment, will be well-formed.

$$\Gamma \vdash e : \tau, \pi \ \wedge\ \vdash E : \Gamma \ \wedge\ \langle E, e\rangle \to C \implies \vdash C : \tau, \Gamma, \pi$$

Together, progress and preservation state that if an expression is well-typed, i.e. the expression is related to a specific type by the typing relation, then evaluation of the expression will not fail and the result will be well-formed w.r.t. to the specific type. This gives that the result of the analysis is a sound approximation of the program semantics.

□

1.2 Usage Analysis

The particular type-based program analysis considered in Paper I of this thesis is a *usage analysis* [LGH+92, Mar93, TWM95, Gus98, WPJ99, WPJ00, GS00, Wan02].

Usage analysis analyzes programs written in lazy functional languages and is best explained by describing how lazy evaluation works. The main feature of lazy evaluation is that an expression is not evaluated before it is needed and is only evaluated once, i.e., its value is shared by all its successive uses. Consider the following example program.

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

When evaluating the program, the expressions bound to `x` and `y` will be stored in an unevaluated form in the program memory, often referred to as the *heap*. We refer to unevaluated 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 it has been evaluated we make sure that we update (replace) the closure with the result. In this way we make sure that when `x` is used a second time, we use the already computed result stored on the heap. It is important to note that since `y` was not needed to evaluate `x + x`, its value was never evaluated.

Lazy evaluation allows the programmer to focus more on what should be computed instead of in which order the computation should be done. It also allows for a natural use of infinite structures such as infinite lists which is more complicated in languages with strict evaluation semantics.

One inefficiency of lazy evaluation is that updating evaluated closures is not always needed. If we change the expression in the example above to `x + x + y` instead of `x + x`, the unnecessary overhead is revealed. When evaluating this expression, `y` will only be used once which means that the time spent updating its closure is unnecessary. Whether this is a serious problem or not depends, of course, on how common expressions that are only used once are in typical 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, which

would reduce the running time. Besides avoiding updates, this information can also be used to enable a number of optimizing program transformations such as inlining, let-floating, and full laziness [PJPS96].

The result of the usage analysis is given in the form of an annotated version of the analyzed program. Each point in the annotated program that allocates a closure is annotated with 1 or ω. The annotation 1 means that all closures created at that point will at most be used once and the annotation ω means that the closures could potentially be used more than once. The following is the result of annotating the modified program.

```
let  x ≙^ω 1 + 2
     y ≙^1 3 + 4
in   x + x + y
```

Subtyping

Two of the features investigated in Paper I are subtyping and polymorphism. Subtyping is useful to allow for a degree of separation between the types of a variable at its binding point and its use points.

The usage analysis associates each variable with a type annotated with usage annotations. For the example above, `x` will be give the type Int^{ω} and `y` the type Int^{1}.

When typing, for example, lists and branches, all elements and subexpressions must be of the same type. Consider the following example where `x` and `y` are inserted in a list.

```
let  x = 1 + 2
     y = sq x
in   [x, y]
```

If the variables are forced to have the same type at their binding point and their use points, `x` will have the type Int^{ω} at its use point because it is used more than once. Since `x` and `y` are elements of the same list, they are forced to have the same type which means that the binding point of `y` will also have the type Int^{ω}. This is undesirable since `y` is only used once and we would, therefore, want its binding point to have the type Int^{1}.

A solution to this problem is to add subtyping. In essence, subtyping allows a type to be seen as a larger type whenever it is semantically safe. By using subtyping we can give `y` the type Int^{1} and choose to see the type of `x` as Int^{1} at its use point. In this way the variables can be seen as having the same type when inserted into the list and we do not break the constraint that all elements of a list should have the same type. This is safe to do since a variable that is annotated as being used more than once may safely be used as a variable that is only used once. The opposite does, however, not hold.

Polymorphism

Polymorphism is needed to express dependencies between usage annotations within a type. Consider typing the identity function:

$$\texttt{id} = \lambda \texttt{x}.\ \texttt{x}$$

What type should it be given? If we give it the type $\forall\alpha.\ \alpha^1 \to \alpha^1$ it will not be possible to use its result more than once. We cannot give it the type $\forall\alpha.\ \alpha^\omega \to \alpha^\omega$ either since it would force all expressions it is applied to, to be annotated with ω.

We want to give it a type that reflects that the usage of its argument will be the same as the usage of its result. This is made possible by the introduction of usage variables, ranged over by k, which can be universally quantified. This corresponds to adding polymorphism to the annotation language and allows us to express the dependence by giving `id` the type $\forall\alpha, k.\ \alpha^k \to \alpha^k$. In Paper I, we investigate a number of different types of annotation polymorphism and evaluate the impact they have on the precision.

2 Program Verification

In this thesis, program verification refers to deductive verification based on logic and theorem proving [RV01, BHS07]. In a program verifier of this type, both the semantics of the program language and the semantic properties to be established are expressed using a program logic. The properties are established by constructing a proof using the calculus of the verifier.

This type of program verification is a more precise and general purpose technique than program analysis. It is more precise because it can encode the exact semantics of the program language in its program logic and perform the reasoning using the logic in a less approximate way. It is more general purpose since it can handle all properties that can be expressed in its logic, which stands in contrast to program analyses which are, typically, tailor-made for a specific property.

This is, however, associated with a considerable cost. Having an expressive program logic means that it is impossible to find a complete calculus. A consequence of this is that it will not be possible to establish 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 both very time consuming and expensive. It also requires the user to have a good knowledge of how the underlying program logic and calculus work.

Deductive program verification does often not have any termination guarantee. When trying to verify an undecidable property, a program verifier will in general be unable to prove or refute the property and, thus, never terminate. It is also the case that since a program verifier is typically made to be very general, it will not be optimized for any special class of semantic properties. There

will, therefore, often be a rather large overhead when establishing properties using the program logic instead of using program analyses tailor-made for the particular properties.

This leaves room for improvement. A typical situation when constructing a proof of a property is that a number of simpler properties have to be established. For some of these, the precision is not crucial and they could well be established by using an approximate program analysis. Examples of such properties include *null-pointer information* [NNH99] and *dependence information* [Wol89].

In Paper II of this thesis we show how a particular program analysis, *reaching definitions analysis* [NNH99], can be embedded in the tactic language of a theorem prover for JAVA. This allows for a close integration of theorem proving and program analysis. The embedding of the particular analysis is, however, only made as proof of concept and the embedded analysis is not used by the theorem prover. In Paper III we go a step further and give a concrete example of how the theorem prover can interact with a program analysis by showing how the handling of loops can be made more automatic by the use of a *dependence analysis*.

The KeY System The program verifier used in Paper II and Paper III is the KeY system [BHS07] which features an interactive theorem prover for formal verification of sequential JAVA programs. In KeY the program to be verified and the properties to be established are modeled in a dynamic logic called JAVA DL [Bec01]. JAVA DL is a modal logic in which JAVA programs occur as parts of formulas using modality operators. The formula $\langle \mathtt{p} \rangle \, \phi$ expresses that the program `p` terminates, without throwing any exceptions, in a state in which ϕ holds. 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 the postcondition ψ holds in the terminating state. Deduction in the JAVA DL sequent calculus is based on symbolic execution and transformation of the programs occurring in the formulas.

The rules of the calculus, called taclets, are implemented in a domain specific tactic language and typically consist of a guard pattern that is matched against the sequent under consideration and an action which is performed if the taclet is applied. In Paper II we show that this language is sufficient to be able to implement the rules a of program analysis.

3 Included Papers

Paper I: Polymorphism, Subtyping, Whole Program Analysis and Accurate Data Types in Usage Analysis In this paper we study what impact a number of features have on the precision of a full scale implementation of a usage analysis for Haskell. The features we investigate are *polymorphism*, *subtyping*, *datatype annotation* and *whole program analysis*.

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. The results of this paper provide guidance to this.

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

- different degrees of polymorphism:
  - constrained polymorphism with polymorphic recursion
  - constrained polymorphism with monomorphic recursion
  - the simple polymorphism introduced by [WPJ00]
  - complete monomorphism
- with and without subtyping
- different treatments of data types
  - full precision (no limit on the number of annotation variables)
  - limit of 100, 10 and 1 annotation variable(s)
  - no annotation variables
- 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.

This is an extended version of the paper [GGS06], published in the proceedings of the Fourth ASIAN Symposium on Programming Languages and Systems 2006 (APLAS'06).

Paper II: Embedding Static Analysis into Tableaux and Sequent based Frameworks In this paper we present a method for embedding program analysis into tableaux and sequent based frameworks. The reason for doing this is to strengthen them by providing access to automatic program analyses.

In these frameworks, the information flows from the root node to the leaf nodes which makes it hard to synthesize information. A major contribution of this paper is that 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.

By using this bi-directional flow of information we are able to implement the rules of a reaching definitions analysis using the taclet language of the KeY system. This allows for a close integration of theorem proving and program analysis. Although the embedding of the particular analysis is only given as proof of concept and is not used to guide the theorem prover, it opens up for new areas of application of tableaux and sequent based theorem provers.

This is a modified version of the paper [Ged05], published in the proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods 2005 (TABLEAUX'05).

Paper III: Verification by Parallelization of Parametric Code In this paper we show how interactive proof techniques, such as induction, can be replaced with automated first-order reasoning in order to deal with parallelizable loops, where a loop can be parallelized whenever the loop iterations are independent of 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.

This is a typical example of where a theorem prover can benefit from using program analysis. In this paper we develop a dependence analysis that ensures parallelizability of loops. When verifying a loop, the theorem prover runs the dependence analysis and uses the result to guide the creation of the proof. If the loop is found to be parallelizable, the proof continues by applying a rule that transforms the loop into a universally quantified update of the state change information represented by the loop body. The application of this rule is done automatically and the creation of the proof continues without need for user interaction. If the loop is not found to be parallelizable, the theorem prover falls back on the original techniques. This makes it possible to use automatic first order reasoning techniques to deal with loops.

The method has been implemented in the KeY prover and evaluated with representative case studies from the JAVA CARD domain.

This is a typographically modified version of the paper [GH07], published in Algebraic and Proof-theoretic Aspects of Non-classical Logics 2007, which is an extended version of the paper [GH06], published in the proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning 2006 (LPAR'06).

Paper IV: Abstract Interpretation Plugins for Type Systems In this paper we show how type-based program analyses can be strengthened by getting additional information about the analyzed program's execution from external analyses. One way to make such information available is to integrate supporting analyses computing the information. Such integration is problematic for a number of reasons: 1) it obscures the original intention of the type system, especially if multiple additional analyses are added, 2) it makes use of already available analyses difficult, since they have to be rephrased as type systems, and 3) it is non-modular: changing the supporting analyses implies changing the entire type system.

Using ideas from abstract interpretation we present a method for parameterizing type systems over the results of abstract analyses in such a way that one modular correctness proof is obtained. This is achieved by defining a general format for transferal and use of the information provided by the abstract analyses. The key gain from this method is a clear separation between the correctness of the analyses and the type system, both in the implementation and correctness proof, which allows for an easy way of changing the external analysis and making use of precise, and hence potentially complex analyses.

We exemplify the use of the framework by presenting a parameterized type system that uses additional information to improve the precision of exception types in a small imperative language with arrays.

This is an extended version [GH08a] of the paper [GH08b], published in the proceedings of the 12th International Conference on Algebraic Methodology and Software Technology 2008 (AMAST'08).

Paper V: Plugins for Structural Weakening and Strong Updates In this paper we use the plugin framework presented in Paper IV and present a general way of making use of *may-* and *must-alias* information to achieve flow-sensitive type systems that allow for flow-sensitivity on the heap.

We show how the alias information can be extracted and made available to a type-based program analysis by the use of our plugin framework. We also present two rules that use this information to allow for flow-sensitive heap types: *structural weakening* and *strong updates*.

The rule for structural weakening uses may-alias information to achieve a limited form of flow-sensitivity that allows for type changes on the heap that are compatible with the subtype hierarchy. Basically, this rule allows us to raise the type of a location, if we at the same type raise the type of all locations aliased with it. This ensures that all aliases have a uniform type view.

The rule for strong updates uses a combination of may-alias and must-alias information and allows for type changes on the heap that are not compatible with the subtype hierarchy, resembling the typical type rule for updates of variables in flow sensitive type systems. This allows us to more freely change the type of locations that have no may-aliases.

This is published as a technical report 2008 [GH08c].

4 Personal Contributions

Paper I: Polymorphism, Subtyping, Whole Program Analysis and Accurate Data Types in Usage Analysis
I contributed to the design of the usage analyses' handling of data types and the specific features of the Core language. I implemented the analyses, integrated them in GHC, modified the runtime system of GHC to compute the necessary statistics and carried out all measurements.

Paper II: Embedding Static Analysis into Tableaux and Sequent based Frameworks
I am the sole author of this paper.

Paper III: Verification by Parallelization of Parametric Code
This paper was written together with Reiner Hähnle. I developed most of the technical material and also made the prototype implementation.

Paper IV: Abstract Interpretation Plugins for Type Systems
This paper was written together with Daniel Hedin. We contributed equally to the technical material.

Paper V: Plugins for Structural Weakening and Strong Updates
This paper was written together with Daniel Hedin. We contributed equally to the technical material.

Bibliography

[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.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[CC77a] Patrick Causot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 77–94, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[Coo85] Keith D. Cooper. Analyzing aliases of reference formal parameters. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 281–290, New York, NY, USA, 1985. ACM.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310, New York, NY, USA, 1990. ACM.

[Ged05] Tobias Gedell. Embedding static analysis into tableaux and sequent based frameworks. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, Tableaux 2005*, volume 3702 of *LNAI*, pages 108–122. Springer, 2005.

[GGS06] Tobias Gedell, Jörgen Gustavsson, and Josef Svenningsson. Polymorphism, subtyping, whole program analysis and accurate data types in usage analysis. In *Proceedings, The Fourth ASIAN Symposium on Programming Languages and Systems*. Springer, 2006.

[GH06] Tobias Gedell and Reiner Hähnle. Automating verification of loops by parallelization. In *Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, LNAI. Springer, 2006.

[GH07] Tobias Gedell and Reiner Hähnle. Verification by parallelization of parametric code. In *Algebraic and Proof-theoretic Aspects of Non-classical Logics*, LNCS 4460. Springer-Verlag, 2007.

[GH08a] Tobias Gedell and Daniel Hedin. Abstract interpretation plugins for type systems. Technical Report 2008:10, Computing Science Department, Chalmers, 2008.

[GH08b] Tobias Gedell and Daniel Hedin. Abstract interpretation plugins for type systems. In *Proceedings, 12th International Conference on Algebraic Methodology and Software Technology*, LNCS. Springer, 2008.

[GH08c] Tobias Gedell and Daniel Hedin. Plugins for structural weakening and strong updates. Technical Report 2008:13, Computing Science Department, Chalmers, 2008.

[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.

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

[Pie02] Benjamin C. Pierce, editor. *Types and Programming Languages*. MIT Press, 2002.

[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.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[RV01] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[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.

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

[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.

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

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 *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:

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

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:

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

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*.

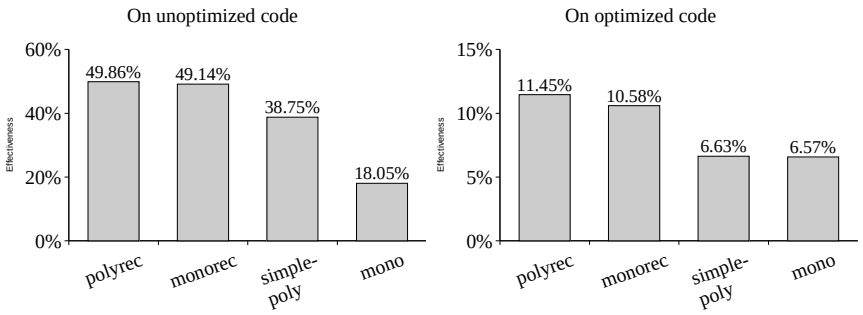


Figure 1: Measurements of polymorphism

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

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

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

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

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

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

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

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

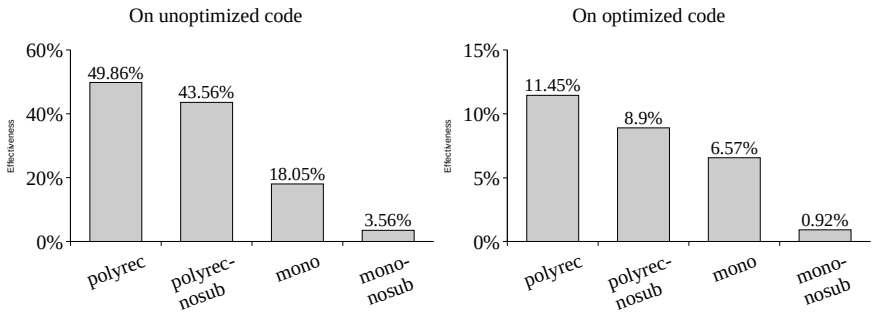


Figure 2: Measurements of subtyping

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.

$$data\ T\ \vec{\alpha} = C_1\ \vec{\tau}_1 \mid \ldots \mid C_n\ \vec{\tau}_n$$

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.

$$
\begin{aligned}
&data\, T_0\, \langle k_0 \rangle \;= C\; Int^{k_0} \\
&data\, T_1\, \langle k_0, k_1, k_2, k_3 \rangle = C_1'\, (T_0\, \langle k_1 \rangle)^{k_0} \mid C_1''\, (T_0\, \langle k_3 \rangle)^{k_2} \\
&\ldots \\
&data\, T_n\, \langle k_0, \ldots, k_m \rangle = C_n'\, (T_{n-1}\, \langle \ldots \rangle)^{k_0} \mid C_n''\, (T_{n-1}\, \langle \ldots \rangle)^{k_{m/2}}
\end{aligned}
$$

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.

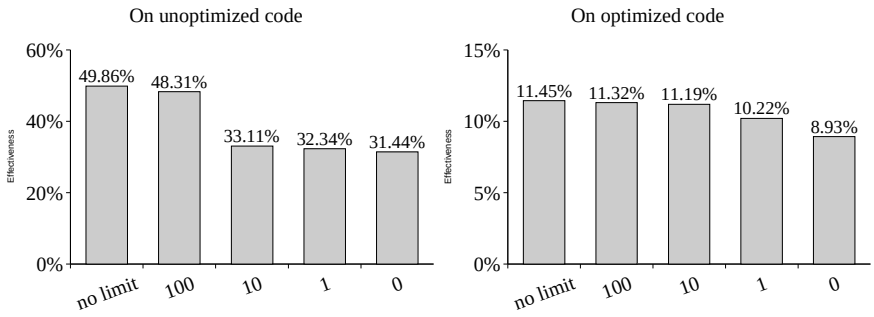


Figure 3: Measurements of treatments of data types

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.

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

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

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

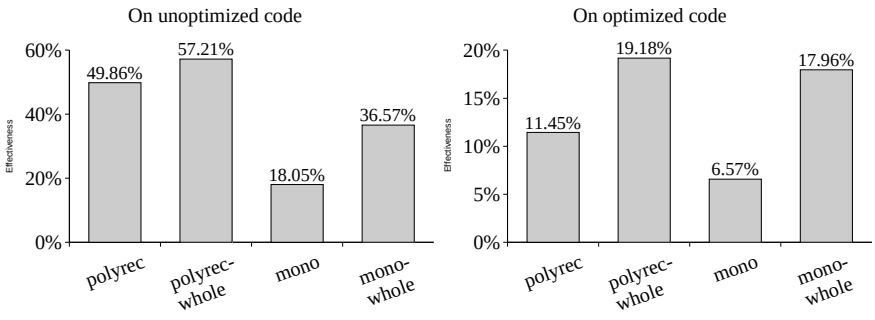


Figure 4: Measurements of whole program analysis

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 this work is to integrate static program analysis with theorem provers used for program verification. In order to do so, the mismatch between the synthetic nature of static program analysis and analytic nature of tableaux and sequent calculi must be bridged. 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, computing some information for each component and merging the computed 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 exchange of information between different branches. Neither is there a need for this, since the rules of the calculus only extend the proof by adding new nodes and adding a node to a branch has no effect on the other branches. Because of this, the information flow in a ground calculus is uni-directional—directed top-down, from the root to the leaves 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]. The addition of free variables breaks the uni-directional flow of information. When a branch instantiates a free variable, the instantiation has to be propagated to the point where the free variable was introduced and all points where it is used. Therefore, there must exist a flow of information going backwards in the proof tree. 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 doing so is that logics for program verification could greatly benefit from an integration with program analysis. An example of this is the verification of loops. When dealing with recursion, user interaction is often needed, which makes the verification very costly. Having access to cheap program analyses could reduce this need for interaction and, thus, reduce the overall cost. It can also reduce the cost of verifying program constructs that the verifiers can cope with automatically. This is the case since program analyses are often tailor made for specific properties and are, thus, often much more efficient than a general purpose verifier.

The following are the main contributions of this work.

- 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 by presenting the full implementation of a program analysis in an interactive theorem prover.

The outline of this paper is as follows. In Section 2, we discuss the bi-directional flow of information introduced by free variables. In Section 3, we briefly describe the used theorem prover. Section 4 and Section 5 present the used program analysis and its implementation in the theorem prover. In Section 6, we draw conclusions and in Section 7, we discuss future work.

2 Flow of Information

By using the mechanism of free variables, information can be exchanged between arbitrary nodes in a proof. This is very useful since our program analysis propagates information computed at the subcomponents of the program to the root node. In a proof, the subcomponents of the program correspond to leaf nodes. To illustrate 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 express a number of analyses. One example is an analysis that establishes whether a property P holds for any branch in a proof. In order to do this, the rule for closure is modified. Normally, the closure rule finds two formulas φ and $\neg\psi$ occurring in the same branch and a substitution that unifies φ and ψ. The closure rule is modified to find a closing substitution for a branch and check whether P holds. If it does, 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 check whether I has been instantiated to c. If it has, we know that P holds for at least one of the branches.

There is still a limit to how much information that can be passed to the root node. It is not possible to gather unique information from each branch since they all instantiate the same variable, I. This can be changed by modifying the extension rule in the following way. When two branches are introduced in a proof, two new free variables, I_L and I_R, are introduced and I is instantiated to $branch(I_L, I_R)$. I_L is used for the leftmost branch and I_R for the rightmost branch. This ensures that each branch instantiates a unique variable, and removes the possibility of conflicting instantiations, since each variable is 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, the instantiation of I will be the term $branch(branch(info_1, info_2), branch(info_3, info_4))$ which contains the information of all four branches. The tableau calculus has, thus, been used to synthesize information from the leaf nodes.

This is almost enough to be 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 solve this problem by, instead of always using the symbol *branch*, allowing for arbitrary function symbols.

2.1 Non Destructive Calculi

In a non destructive constraint tableau, as described in [Gie01], we can express 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 propagated to the sink object of the parent. If the parent 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 its children. If it is, then it is propagated upwards, otherwise it is discarded. If the parent has only one child, the substitution is propagated upwards directly.

A tableau working like this is called non destructive since the free variables are never destructively instantiated. Instead, a set of all possible closing instan-

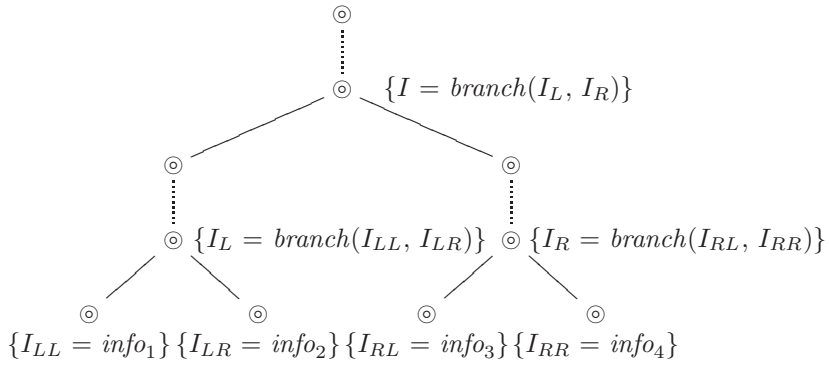


Figure 1: Example Tableau

tiations is computed 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, give us the result that the tableau is closable but also give us the closing substitution. The mechanism of the sink objects can also make it easier to implement some of the extensions described in Section 7.

3 The KeY Prover

The theorem prover used to implement our program analysis is the KeY system [ABB+05]. The KeY system is an interactive 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 following formula which denotes that after executing the assignment `i = 1` the value of the variable `i` is greater than 0.

```
<{ i = 1; }> i > 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 have a large set of rules at hand to handle all possible program constructs. Instead of hard-wiring these

into the core of the theorem prover, a more general solution is to implement the rules using a domain specific tactic language.

The KeY system has such a tactic language and the rules written in this language are called taclets [BGH+04]. 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 a pattern to match a sequent is that there is a substitution, unifying 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 branches.

To introduce the syntax of the KeY tactic language, we present one of the simplest rules, the `close_by_true` rule.

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

The pattern matches sequents where *true* is found on the right-hand side. If it is, 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, as shown by the following rule.

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

The rule replaces negated formulas on the left-hand side, by their negation on the right-hand side. The proof branch remains open, but contains the modified sequent. New branches can be created by having multiple `replacewith` actions.

So far, we have only considered sequents that do not contain embedded programs. When embedding programs in formulas, a modality operator is used. There are a number of different modality operators having different semantics. The diamond operator `<{p}>`ϕ expresses that there is a terminating execution of the program `p`, after which the formula ϕ holds. The box operator `[{p}]`ϕ expresses that after all terminating executions, the formula ϕ holds. For our purposes, the modalities do not have any meaning since we are not trying to construct a proof in the traditional way. The syntax of the taclet language does, however, force 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 can be used in 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 implement using our technique is a *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 approximates the set of assignments that may reach each individual statement in a program. Consider the following program, consisting of three assignments, where each statement is annotated with a label.

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

Consider the statement annotated with 1. The statement executed before it (which we will refer to as its *preceding* statement) is the assignment $\texttt{a} \overset{0}{=} \texttt{1}$ and since `a` has not been modified, it still contains the value 1. We say that the assignment annotated with 0 *reaches* the statement annotated with 1.

For each statement, we compute 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, respectively. In 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 modified. We do not only store the labels of the assignments, but also the names of the assigned variables. The following are the entry and exit sets of our example program.

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* since the reaching definitions problem is undecidable. We do, however, ensure that the approximation is *sound*, 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 opposite may not hold, a label of an assignment being present in an entry set of a statement, does not necessarily mean that the assignment actually reaches 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, however, not be useful.

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

4.1 Input Language

The input language is a simple while-language, consisting of assignments, block statements and if- and while-statements. We use a simple language because we do not want to wrestle with a large language but instead show the concept of how a static program analysis is implemented.

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

lbl ranges over the natural numbers and is 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 consider the constraint-generation part of the analysis and start by defining the collections of equations that are generated. These equations characterize the reaching information in the analyzed program.

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

$\emptyset$ is the empty collection of equations. $\mathbf{Entry}(\mathit{lbl}) = \Sigma$ and $\mathbf{Exit}(\mathit{lbl}) = \Sigma$ define 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 joins two collections of equations.

The set expressions are used to represent entry and exit sets.

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

$\emptyset$ is the empty set (this symbol is overloaded without risk of confusion). $(\mathit{var}, \mathit{lbl})$ is the singleton set consisting of a single reaching assignment. $\mathbf{Entry}(\mathit{lbl})$ and

Exit(*lbl*) refer to 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 the label of the statement executed before s, ℓ_1 the label of the last executed statement in s, and Π the equation characterizing the reaching information of the statement s.

Intuitively, we need the label of the statement executed before s because we use its exit set when analyzing s. We also 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.

In the assignment rule we know that the reaching assignments in the entry set will be exactly those in the exit set of the preceding statement.

Assign

$$\frac{}{\ell_0 \vdash x \stackrel{\ell_1}{=} e; \ \Downarrow \ \ell_1 \ : \ \mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0) \ \wedge \ \mathbf{Exit}(\ell_1) = (x, \ell_1) \cup (\mathbf{Entry}(\ell_1) \ - \bigcup_{\ell \in lbl} (x, \ell))}$$

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 $\mathbf{Exit}(\ell_1) = (x, \ell_1) \cup (\mathbf{Entry}(\ell_1) \ - \bigcup_{\ell \in lbl}(x, \ell))$.

The rule for if-statements is defined as follows.

If

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

The entry set is equal to the exit set of the preceding statement, which is expressed by the equation $\mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0)$. When analyzing the branches s_0 and s_1, we use l_0 as the label of the preceding statement since it is important that they, when referring to the exit set of the preceding statement, use $\mathbf{Exit}(l_0)$ and not the exit set of the if-statement.

From the branches, we get the collections of 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. Since we do not know which branch is going to be taken, we 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 sets of the branches, expressed by the equation $\mathbf{Exit}(\ell_1) = \mathbf{Exit}(\ell_2) \cup \mathbf{Exit}(\ell_3)$.

The rule for while-statements is defined as follows.

While

$$\frac{\ell_1 \vdash s \ \Downarrow \ \ell_2 \ : \ \Pi_0}{\ell_0 \vdash \mathbf{while}_{\ell_1}(e) \ s \ \Downarrow \ \ell_1 \ : \ \Pi_0 \ \wedge \ \mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0) \cup \mathbf{Exit}(\ell_2) \ \wedge \ \mathbf{Exit}(\ell_1) = \mathbf{Entry}(\ell_1)}$$

For the entry set, we include the exit set of the preceding statement, but also the exit set of the last executed statement in the loop body. We 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, therefore, want the exit set to be the union of the entry set of the while-statement and the exit set of the last executed statement in s. Since this is the exact definition of the entry set, we let them be equal.

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 is the statement last executed. The second time, and all times after that, it should use the exit set of l_1, since the body of the while loop is the statement last executed. We approximate this by not separating the two cases and always use l_1 as the label of the preceding statement.

We do not have a special rule for programs. Instead, we treat a program as a block statement and use the following rules for sequential statements.

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

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

5 Embedding the Analysis into the Prover

5.1 Encoding the Datatypes

In order to encode Σ, Π, and labels, we need to declare the used types. 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 in the KeY system.

In the constructors for Σ, defined by (2), we have, for convenience, replaced the difference operator with the constructor **CutVar**, where **CutVar**(s, x) denotes the set expression $s - \bigcup_{\ell \in lbl}(x, \ell)$. The 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 the statements ourselves. In order to generate labels we define **Zero** and **Succ**, with which we can easily enumerate all needed labels. The first label is **Zero**, the second **Succ(Zero)**, and so on.

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

Since the rules of the analysis use the exit set of the preceding statement, the very first statement of a program (which does not have any preceding statement) requires special treatment. We define the label **Start** which is exclusively used as the label of the (non-existing) statement preceding the first statement. When solving the equations we let the exit set of this label 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. We wrap the label of the preceding 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 is attached to it). In the predicate, we also include two labels, needed for the generation of the labels used to annotate the statements. The included labels are the first unused label before annotating the statement and the first unused label after annotating 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 as follows.

```
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;
```

The rules for empty block statements is implemented as a taclet matching 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 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 defined 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 involved. 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 is defined as follows, where we let $\bar{s_2}$ range over lists of statements.

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

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, `.. ...`,[1] match the tail. We 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 in 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 increment the counter of the first free label. We do that by letting the fifth argument be the successor of `lbl1` (the fifth argument in the wrapper predicate is the first free label after annotated the statement). Because of implementation details in KeY, we need to use the **varcond** construction to bind `name` to the name of the variable matching `#x`.

```
rdef_assign  {
 find (==> <{#x = #e;}>
```

[1] The leading two dots match the surrounding context, which for our analysis is known to always be empty.

```
  (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
};
```

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 computed the correct entry and exit sets, which is not 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 one 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`, giving us the following.

```
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 is 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)
```

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

When the proof has been created, we fetch the instantiations 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)
}
```

These constraints are solved by a stand-alone constraint solver. Recall that the analysis consists of 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 form of the formula or statement under consideration. 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 strange that we can 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 mechanism of meta variables, that we use to create a bi-directional flow of information. Using meta variables, we can synthesize almost any type of information. We are, however, limited in what computation we can do on the information. So far, we cannot do any computation 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 of KeY 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

The work presented in this paper is a starting point and opens up for a lot of future work including the following.

- Try different theorem provers to see how well the method presented in this paper works in combination with other theorem provers.
- Further analyze the overhead of using a theorem prover to implement program analyses.
- Modify the calculus of the KeY system to make use of the information computed 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 achieved.
- Investigate other analyses. We implemented 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 analyses that would benefit the KeY system. Among the possible candidates for this are:
  - An analysis that computes 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 computing 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.

A limitation of the sequent calculus in the KeY system 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 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 and difference. In a constraint tableaux setting, the simplification of these operations could 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 to what degree static program analysis and theorem proving should be integrated. The level of integration can vary from having a program analysis analyzing a program and passing the results 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.

Verification by Parallelization of Parametric Code

Tobias Gedell Reiner Hähnle

Abstract

Loops and other unbound control structures constitute a major bottleneck in formal software verification, because correctness proofs over such control structures generally require user interaction: typically, induction hypotheses or invariants must be found or modified by hand. Such interaction 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. A loop can be parallelized, whenever the execution of a generic iteration of its body depends only on the step parameter and not on other iterations. We use a symbolic dependence analysis that ensures parallelizability. This guarantees soundness of a proof rule that transforms a loop into a universally quantified update of the state change information effected by the loop body. This rule makes it possible to employ automatic first-order reasoning techniques to deal with loops. The method has been implemented in the KeY verification tool. We evaluated its applicability with representative case studies from the JAVA CARD domain.

1 Introduction

The context of this paper is formal software verification of object-oriented programs. The target programs are executable JAVA programs (not abstract programs) and we want to prove complex functional properties of these. There are a number of software verification systems that target JAVA and related programming languages [BLS05, BHS07, BRL03, MPM05, PHM99, Ste05]. All of these systems are semi-automatic at best. The reason is that the emergence of undecidable predicates is typical when proving correctness for the combination of data structures of unbounded size and of control structures that can lead to an unbounded number of execution steps. Typical examples of the former include integers, lists (arrays), trees. The most important representatives of the latter are loops, recursive method calls, and concurrent processes. All of them are present in JAVA-like languages.

If we do not want to abstract away from real JAVA programs as in software model checking [Hol02] or trade off verification for mere bug finding [FLL+02], then the inherent limitations of computability do not allow a complete, uniform deduction system for program verification. Even though it seems that the calculi used for program verification are practically complete in the sense that complex,

realistic examples can be handled [BHS07, Bre06, JMR04] without encountering incompleteness phenomena, this is not enough. To see why, let us look at an example.

Example 1.1 (Array reversal)
The following loop reverses the elements of the `int` *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;
}
```

A formal specification can be given in first-order logic as follows:

Precondition: $\texttt{a} \neq \texttt{null}$
Postcondition: $\forall j.(0 \leq j < \texttt{a.length} \rightarrow \texttt{a}[j] \doteq \texttt{\textbackslash old}(\texttt{a})[\texttt{a.length} - 1 - j])$

The keyword `\old` *indicates that the value* `a` *had before the execution is referred to.* □

What are the options to prove total correctness of this loop with respect to its contract? Finite unwinding is impossible and abstraction has difficulties to record that the value `a.length` depends on `a`. The standard approach is to use one of two general-purpose mechanisms for dealing with unbounded control structures, *invariants* or *induction*. In the first case, one would establish that the loop preserves a suitable invariant property I, which must be strong enough to imply the postcondition. Termination of the loop is proven separately (and is trivial for this example). Alternatively, an induction argument over `i` would typically establish that the loop reverses all array positions. The problem is that both, a suitable invariant and a suitable induction hypothesis, are not straightforward to derive from the postcondition: it is necessary to introduce a new variable k for the index up to which the array has been reversed already, k must have appropriate bounds, the precondition must be included, etc. In general, the postcondition might not be given (for example, if the task is to derive the specification from the code). In this case, it is even more difficult to come up with a suitable invariant or induction hypothesis. In addition, loop rules in realistic imperative languages [BSS05] are very complex. User interaction involves a high amount of technical knowledge and is thus extremely expensive.

There is a large body of work on heuristically guided inductive theorem proving, but most of it is done in the context of functional programming [BJ88, BBHI05]. Existing work on automatic synthesis of loop invariants in imperative programs [LL05, RCK05] is defined only for an abstract while-language. A recent divide-and-conquer technique for decomposition of induction proofs [OW05] works for a larger fragment of JAVA, but it is targeted at simplifying user interaction rather than eliminating it.

Main Contributions

The contribution of this paper is to present a new verification technique that relies neither on abstraction, nor on invariants, nor on induction. It is *complementary* to the work cited above in so far as our goal is to recognize situations where complex invariants can be avoided altogether.

The key insight (illustrated by means of Example 1.1) is that the swap operations realized in the loop body can be executed independently of each other: the assignment to `a[i]` and the value of `a[a.length - 1 - i]` do not depend on any `a[j]` and `a[a.length - 1 - j]` with $\texttt{i} \neq \texttt{j}$ provided that `i` and `j` are within the bounds specified in the guard of the loop.

Now, a new way to prove correctness of the loop goes as follows: first compute the effect of a generic iteration of the loop body parameterized with `i`; second, prove that there are no dependencies between different iterations in the loop range; third, generalize the effect of the loop body over all values that the parameter `i` takes on in the loop range; and fourth, prove that the postcondition is implied by the loop. Importantly, the last step involves no induction, but automatable first-order search stratagems such as quantifier elimination and term rewriting.

Obviously, verification by parallelization of parametric code is an *incomplete* verification technique for loops, because not all loops are parallelizable. On the other hand, it is not an exotic special case either: from an analysis of the *unchanged* code of several real JAVA CARD programs we concluded that parallelizable loops occur naturally and relatively frequently, see Section 10. As we show in Section 11, verification by parallelization is not restricted to loops, but can be applied whenever a non-linear program is composed of parametric pieces of code, for example, in recursive calls and concurrent processes. In addition, the current trend towards multi-core processors will result in more code being written in such a way that it is parallelizable. Therefore, verification by parallelization is a *relevant* technique for increasing the degree of automation in software verification.

The most important aspect of verification by parallelization is that it is a *highly automatable* verification technique. First, because the computation of the effect (i.e., the strongest postcondition relative to a given precondition) $\mathcal{U}(\texttt{i})$ of a piece of code `p(i)` parameterized by `i` is done automatically. Even the choice of the parameter `i` is automatic and guided by heuristics. The details are given in Sections 4 and 5. Second, the effect of some non-linear parameterized code (such as a loop with body `p(i)`) is represented in form of a universally quantified state update, say, $\texttt{\textbackslash for int } I; \{\texttt{i} := I\}\{\mathcal{U}(\texttt{i})\}$. Therefore, it can be further processed during the remaining verification proof by employing first-order reasoning, see Section 8.

Soundness of the universal quantification step is ensured by an automatic symbolic dependence analysis described in Sections 6 and 7. This analysis is executed not directly on the code `p(i)`, but on the simplified and normalized effect $\mathcal{U}(\texttt{i})$ computed by symbolic execution before. This feature makes our approach *robust*, because the success of the dependence analysis does not rely

on any syntactic restrictions of `p(i)`. A further robustness feature is that our dependence analysis does not simply fail in case when dependencies are detected, but yields a symbolic constraint that is sufficient for dependencies not to occur and that can be used elsewhere in the verification attempt, see Section 9.

In the following section, we collect a number of technical notions needed later on. 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 [BHS07], 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 fully axiomatizes the JAVA CARD programming language [Jav03] which contains all JAVA features minus multi-threading, floating point types, and dynamic class loading. It has also some features that JAVA does not have, but they are not addressed in this article.

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 occurring in JAVA DL formulas are 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 the 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 without causing excessive branching. In the JAVA DL calculus a different solution is used, based on the notion of (state) *updates*.

Definition 2.1 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* $\{$`loc`$:=$ `val`$\}\phi$ *is the same as that of* $\langle$`loc=val;`$\rangle\,\phi$*.*

Changes of the computation state can be represented with the help of updates. For example, the update $\{$`loc`$:=$ `val`$\}\phi$ represents all states in which the formula ϕ holds after the value of `loc` has been changed to `val`. In a somewhat loose manner we use updates to represent states, for example, the update $\{$`loc`$:=$ `val`$\}$ is used to represent an arbitrary state, where the value of `loc` is `val`.

Definition 2.2 *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)* \if (b) $\{\mathcal{U}\}$*, where* b *is a quantifier-free formula (*conditional execution*), (iv)* \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*), (v)* $\{\mathcal{U}\}\mathcal{U}'$ application*.*

The semantics of sequential, conditional, and application 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 \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*[1] *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 [Rüm06], see also [BHS07].

Sequential composition of updates is automatically transformed into parallel composition in KeY and we will therefore mostly not consider it further.

3 Outline of the Approach

Let us look at the following example:

[1] Well-orders are usually defined with respect to minimal elements. We use the dual definition here, because it is more natural in our setting.

```
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 $\mathcal{S} = \{\texttt{i} := 1\}$. 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, $\texttt{c} := 1$, we would start the execution in the state $\{\texttt{c} := 1\}$ and the resulting state would be much simplified.

In our example, symbolic execution of one loop iteration starting in the empty state gives $\mathcal{S}' = \{\texttt{i} := \texttt{i} + 1,\ \texttt{\textbackslash if}\ (c \neq 0)\ \{\texttt{a[i]} := \texttt{b[i+1]}\},\ \texttt{\textbackslash if}\ (c \doteq 0)\ \{\texttt{a[i]} := \texttt{b[i-1]}\}\}$, 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 and Section 7. In the example it results in a logical constraint $\mathcal{C}$ that, among other things, contains the disjunction of $\texttt{c} \neq 0$ and $\texttt{a} \neq \texttt{b}$. 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, $\texttt{b.length} > \texttt{a.length}$.

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 $\{\texttt{c} := 1\}$ and the termination constraint in $\mathcal{D}$ holds. In this case, we can additionally simplify $\mathcal{S}'$ to $\{\texttt{c} := 1,\ \texttt{i} := \texttt{i} + 1,\ \texttt{a[i]} := \texttt{b[i+1]}\}$.

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 8.

The result for the example in somewhat simplified manner is as follows:

$$\begin{array}{l}\texttt{\textbackslash for int } n;\\ \quad \{\texttt{i} := n+1\}\\ \quad \{\texttt{\textbackslash if } (\texttt{i} \geq 1 \wedge \texttt{i} < \texttt{a.length})\ \{\texttt{c} := 1,\ \texttt{i} := \texttt{i}+1,\ \texttt{a[i]} := \texttt{b[i+1]}\}\}\end{array}$$

Here we make use of an update applied to an update. The variable n holds the iteration number, i.e., 0 for the first iteration, 1 for the second, and so on. For each iteration we need to assign the loop variable its value. This is done by the update $\{\texttt{i} := n+1\}$. We apply this update to the guarded update which has the effect that all occurrences of `i` in non-update positions (guard, arguments, right-hand sides) are replaced by $n+1$. The resulting update is:

$$\begin{array}{l}\texttt{\textbackslash for int } n;\\ \quad \{\texttt{\textbackslash if } (n+1 \geq 1 \wedge n+1 < \texttt{a.length})\\ \quad \{\texttt{c} := 1,\ \texttt{i} := n+2,\ \texttt{a[}n+1\texttt{]} := \texttt{b[}n+2\texttt{]}\}\}\end{array}$$

The `for`-expression is a universal first-order quantifier whose scope is an update that contains occurrences of the variable n (see Def. 2.2 and [Rüm06]). 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 the Effect of a Generic Loop Iteration

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. Our method handles the fragment of JAVA that the symbolic execution machinery of KeY handles, which is JAVA CARD [Jav03].

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. Complete symbolic execution of `p` starting in state $\mathcal{S}$ eventually yields a proof tree whose open leaves are of the form $\Gamma \rightarrow \mathcal{U}\, \mathit{fin}$ 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 execution paths in the loop body.

Example 4.1 *Consider the following statement* `p`*:*

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

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

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

where V *stands for* $\mathtt{a} \neq \mathtt{null} \wedge i \geq 0 \wedge \mathtt{i} < \mathtt{a.length}$*. We extract the following vectors:*

$$\begin{array}{lcll} \vec{\Gamma} & \equiv & \langle V \wedge i > 2,\ V \wedge i \not> 2 \rangle & \\ \vec{\mathcal{U}} & \equiv & \langle \{\mathtt{a[i]} := 0,\ \mathtt{i} := \mathtt{i} + 1\},\ \{\mathtt{a[i]} := 1,\ \mathtt{i} := \mathtt{i} + 1\} \rangle & (3) \end{array}$$

□

Symbolic execution can become stuck at a leaf containing a program in three ways:

1. The program has been fully executed and only an update and the formula *fin* remain. This is what we call a *success leaf*. The effect of the program was successfully transformed into a state update. Success leaves are always of the form $\Gamma \rightarrow \mathcal{U}\, \mathit{fin}$.

2. Abrupt termination caused by, for example, a thrown exception. In this case the program cannot be transformed into a state update. We call this a *failed leaf.*

3. The strategies for automatic symbolic execution were not strong enough to execute all instructions in the program. This could possibly be remedied by enabling more powerful and expensive strategies and restart symbolic execution. If they are still not strong enough, we count the leaf as a failed leaf.

If a failed leaf can be reached from the initial state, then 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. We construct the vector $\vec{\mathcal{F}}$ consisting of the path conditions Γ of all failed leaves and let the negation of $\vec{\mathcal{F}}$ become a condition that needs to be proven when applying our method.

Example 4.2 *In Example 4.1 we only showed the success leaves. When symbolic execution becomes stuck, there are, in addition to the success leaves, failed leaves of the following form:*

$$\begin{array}{lcl} a \doteq \texttt{null} & \rightarrow & \ldots \text{ fin} \\ a \neq \texttt{null} \wedge i < 0 & \rightarrow & \ldots \text{ fin} \\ a \neq \texttt{null} \wedge i \not< \texttt{a.length} & \rightarrow & \ldots \text{ fin} \end{array}$$

The first leaf corresponds to the case where `a` *is* `null` *and using* `a` *throws a null pointer exception. The second and third leaves correspond to the case where* `i` *is outside* `a`*'s bounds and accessing* `a[i]` *throws an index out of bounds exception. From the failed leaves we extract the following vector:*

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

□

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 4.3 *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 $\mathcal{S}_{init} = \{\mathtt{c} := 1,\ \mathtt{i} := 0\}$*. 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* $\{\mathtt{a[0]} := 0,\ \mathtt{b[0]} := 0,\ \mathtt{i} := 1\}$*, which is the result after the* first *iteration, not a generic one. The problem is that* $\mathcal{S}_{init}$ *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 would get* $\{\backslash\mathtt{if}\ \ (\mathtt{c} \neq 0)\ \{\mathtt{a[i]} := 0\},\ \mathtt{b[i]} := 0,\ \mathtt{i} := \mathtt{i} + 1\}$*, 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 one of the branches we backtrack to state $\mathcal{S}_{\text{init}}$ and use the standard rules to prove the loop. If the guard evaluates to false, we skip the loop and proceed using the standard rules.

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 success leaf. This is done as follows: for each modified location in $\mathcal{S}$, add an update of the location to itself in parallel to the updates in $\mathcal{S}$. They are added syntactically after all updates in $\mathcal{S}$ and, therefore, the clash-semantics of updates ensures that the previous assignments to the modified locations in $\mathcal{S}$ are canceled. More formally, $\mathcal{S}'$ is defined as follows: $\mathcal{S}' = \mathcal{S}, \bigcup_{l \in mod(\vec{\mathcal{U}}, \mathcal{S})} \{l := l\}$, where $mod(\vec{\mathcal{U}}, \mathcal{S})$ is the set of locations in $\mathcal{S}$ whose assigned term in $\vec{\mathcal{U}}$ differs from its assigned term in $\mathcal{S}$. How to compute this set is discussed below.

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 $\mathcal{S}_{\text{iter}}$.

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

Example 4.4 *Example 4.3 yields the following sequence of states:*

Round	*Start state*	*State modifications*	*New state*
1	$\{\texttt{c} := \texttt{1},\ \texttt{i} := \texttt{0}\}$	$\{\texttt{a[0]} := \texttt{0},\ \texttt{b[0]} := \texttt{0},\ \texttt{i} := \texttt{1}\}$	$\{\texttt{c} := \texttt{1},\ \texttt{i} := \texttt{i}\}$[3]
2	$\{\texttt{c} := \texttt{1},\ \texttt{i} := \texttt{i}\}$	$\{\texttt{a[i]} := \texttt{0},\ \texttt{b[i]} := \texttt{0},\ \texttt{i} := \texttt{i+1}\}$	$\{\texttt{c} := \texttt{1},\ \texttt{i} := \texttt{i}\}$

□

Computing the set $mod(\vec{\mathcal{U}}, \mathcal{S})$ 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 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. 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. These issues are handled by using a dependence analysis to compute $mod(\vec{\mathcal{U}}, \mathcal{S})$. The details of how this is done are described in Section 7.

5 Loop Variable and Loop Range

For the dependence analysis and for creating the quantified state update we need to identify a loop variable and the loop range. The requirement we have on a loop variable is that it must, in each success leaf, be updated with the same step function by an unguarded update.

When deciding whether a particular variable `i` is a possible loop variable, we look for a function *step* such that $i := step(i)$ is found in each update $\mathcal{U} \in \vec{\mathcal{U}}$. Remember that $\vec{\mathcal{U}}$ contains the updates from all success leaves. In KeY, finding such a function is often not possible due to eager simplification performed on updates. If, for example, for a specific leaf, the path condition contains $i \doteq 0$ the update $i := i + c$ will be simplified to $i := c$. This means that even if $i := i + c$ is the step function of the loop it will not be found in all leaves. To handle this we must take the path condition Γ into account. For each success leaf with path condition Γ and update $\mathcal{U}$ we require that under the path condition, $step(i)$ is equal to the expression assigned to `i` by $\mathcal{U}$. Formally, this is expressed by $\Gamma \rightarrow step(i) \doteq \mathcal{U}i$.

The step function describes the execution order of the loop iterations and expresses how the loop variable changes between each loop iteration. For constructing the quantified state update we need to know the value that the loop variable has in each iteration of the loop, that is, we need to have a function from the number of an iteration to the value of the loop variable in that iteration. This function is defined as $iter(n) = step^n(start)$ where n is the number of the iteration and *start* the initial value of the loop variable. In JAVA DL we cannot write recursive expressions directly, so we have to rewrite the body of *iter* into a non-recursive expression. This is in general hard, but whenever the loop variable is incremented or decremented with a constant value in each iteration, it is

[3] The new state that gets computed is $\{\texttt{c} := \texttt{1}, \texttt{i} := \texttt{0},\ \texttt{i} := \texttt{i}\}$ but is simplified to $\{\texttt{c} := \texttt{1},\ \texttt{i} := \texttt{i}\}$.

easy to do. At present we impose this as a restriction: the step function must have the form `i + e`, where `i` is the loop variable and `e` is invariant during loop execution. Then one obtains the following definition: $iter(n) = start + n * e$. It would be possible to let the user provide the definition of *iter* allowing for more complicated step functions to be handled. It would, however, come at the price of making the method less automatic.

To identify the loop variable, we start with the set of variables occurring in the loop and remove all those for which a step function cannot be found. After this we might be left with more than one variable. Since we cannot, currently, handle more than one loop variable we need to eliminate the other candidates. If they are not eliminated they would cause data flow-dependencies that could not be handled by our method. A candidate is eliminated by transforming its expression into one 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 + n * c;`, where `init` is the initial value of `l` and `n` the number of the iteration.

To make the identification of loop variables more efficient we use a heuristic that favors variables that occur in the loop guard (as loop variables often do) and that are syntactically small (for example, `i` is considered smaller than `a[l]`).

Example 5.1 *Consider the code in Example 4.1 which gives the vector in (3). The only variable for which a step function can be found is* `i`*. It is, therefore, identified as the loop variable.* □

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 $\Gamma \rightarrow \{$`dummy`$:=$`e`$, \ldots\}$ *fin*, we create the formula GS which characterizes when the guard is true. Formally, GS is defined as $\bigvee_\Gamma(\Gamma \wedge$ `e` $\doteq$ `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 5.2 *Consider the following guard* `g` $\equiv$ `i < a.length`*. When all instructions in the formula* $\langle$`boolean dummy = g;`$\rangle$ fin *have been symbolically executed, there are two success leaves:*

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

From these we extract the following formula GS:

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

After simplification of GS we obtain:

$$a \neq null \wedge i < a.length \quad .$$

When the instructions have been symbolically executed, there is also a failed leaf containing $\mathtt{a} \doteq \mathtt{null} \rightarrow \ldots$ fin. *From it we extract the formula* $GF \equiv \mathtt{a} \doteq \mathtt{null}$. □

The formula GR_n characterizes when the iteration number n is within the loop range. The following definition expresses that there exists an iteration where the loop variable has the value $iter(n)$ and, moreover, this iteration can be reached:

$$GR_n \equiv n \geq 0 \ \wedge \ \{\mathtt{i} := iter(n)\}GS \ \wedge \ \forall m.\ 0 \leq m < n \rightarrow \{\mathtt{i} := iter(m)\}GS$$

It is important that the loop terminates, otherwise, our method would be unsound. We, therefore, create a termination constraint GT that needs to be proven when applying our method. The termination constraint expresses that there exists a number n of iterations after which the guard formula evaluates to false. The constraint GT is defined as:

$$GT \equiv \exists n.\ n \geq 0 \wedge \{\mathtt{i} := iter(n)\}\neg GS$$

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 dependence, 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 6.1 *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 the array `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 6.1 *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 (i.e., appears on the left-hand side of an atomic update) that is read (appears on the right hand side of an atomic update or as an argument 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 6.2 *When executing the second iteration of the following loop, the location* `a[1]`*, which is modified by the first iteration, is read, indicating 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 8 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 6.1) 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. From the step function we extracted the function *iter*, so 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 lower

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.

7 Implementation of the Dependence Analysis

Our dependence analysis analyzes a 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 validity of the generated constraint will then be decided by a theorem prover.

When looking for dependencies in the loop we do not analyze the loop itself but the state updates computed from the generic loop iteration. The dependence analysis is, therefore, defined over updates. The binary function δ defined in Table 1 takes two updates as arguments and computes a constraint that characterizes the absence of data flow-dependence among its arguments. In the definitions, we let *locs(t)* be the set of locations occurring in the term t and *slocs(`loc`)* the set of locations occurring as arguments in `loc` as defined below:

$$\begin{array}{lcl} \mathit{slocs}(\texttt{v}) & = & \emptyset \\ \mathit{slocs}(\texttt{o.f}) & = & \mathit{locs}(o) \\ \mathit{slocs}(\texttt{a[i]}) & = & \mathit{locs}(a) \cup \mathit{locs}(i) \end{array}$$

The computation of the dependence constraint of a loop uses the vectors $\vec{\Gamma}$ and $\vec{\mathcal{U}}$ extracted from the success leaves during symbolic execution of the loop body. They were obtained as the result of a generic loop iteration in Section 4. If the preconditions of two leaves are true for two different loop iterations we need to ensure that the updates of the leaves are data flow-independent of each other (Def. 6.1). Formally, if there exist two distinct iteration numbers k and l and (possibly identical) leaves r and s, for which $k < l$, $\{\texttt{i} := \mathit{iter}(k)\}\Gamma_r$ and $\{\texttt{i} := \mathit{iter}(l)\}\Gamma_s$ are true, then we need to ensure independence of $\mathcal{U}_r$ and $\mathcal{U}_s$.

We do this for all pairs of leaves and define the dependence constraint for

Atomic updates

$$\begin{array}{lcl}
\texttt{v} := \texttt{val}\ \delta\ \texttt{loc} := \texttt{val}' & = & \begin{cases} \texttt{true} & \textbf{when } \texttt{v} \notin (\mathit{locs}(\texttt{val}') \cup \mathit{slocs}(\texttt{loc})) \\ \texttt{false} & \textbf{otherwise} \end{cases} \\
\texttt{o1.f} := \texttt{val}\ \delta\ \texttt{loc} := \texttt{val}' & = & \neg(\bigvee_{o2.f \in (\mathit{locs}(\texttt{val}') \cup \mathit{slocs}(\texttt{loc}))} \texttt{o1} \doteq \texttt{o2}) \\
\texttt{a[i]} := \texttt{val}\ \delta\ \texttt{loc} := \texttt{val}' & = & \neg(\bigvee_{b[j] \in (\mathit{locs}(\texttt{val}') \cup \mathit{slocs}(\texttt{loc}))} (\texttt{a} \doteq \texttt{b} \wedge \texttt{i} \doteq \texttt{j}))
\end{array}$$

General updates

$$\begin{array}{lcl}
\mathcal{U}\ \delta\ \backslash\texttt{if}\ \ (b)\ \{\mathcal{U}'\} & = & \neg b \vee \mathcal{U}\ \delta\ \mathcal{U}' \\
\backslash\texttt{if}\ \ (b)\ \{\mathcal{U}\}\ \delta\ \mathcal{U}' & = & \neg b \vee \mathcal{U}\ \delta\ \mathcal{U}' \\
\mathcal{U}\ \delta\ \backslash\texttt{for}\ T\ s; \mathcal{U}'(s) & = & \forall s.\ \mathcal{U}\ \delta\ \mathcal{U}'(s) \\
\backslash\texttt{for}\ T\ s; \mathcal{U}(s)\ \delta\ \mathcal{U}' & = & \forall s.\ \mathcal{U}(s)\ \delta\ \mathcal{U}' \\
\mathcal{U}_0, \ldots, \mathcal{U}_m\ \delta\ \mathcal{U}'_0, \ldots, \mathcal{U}'_n & = & \bigwedge_{i,j} \mathcal{U}_i\ \delta \mathcal{U}'_j
\end{array}$$

Table 1: Computing dependence constraints among updates.

the entire loop as follows where GR is the loop range predicate and $\mathcal{I}_{r,s,k,l}$ is defined as $\{\texttt{i} := \mathit{iter}(k)\}\mathcal{U}_r\ \delta\ \{\texttt{i} := \mathit{iter}(l)\}\mathcal{U}_s$.

$$\mathcal{C} \equiv \bigwedge_{r,s} \forall k, l.\ \left(k < l \wedge \left(\begin{array}{c} GR_k \wedge GR_l\ \wedge \\ \{\texttt{i} := \mathit{iter}(k)\}\Gamma_r \wedge \{\texttt{i} := \mathit{iter}(l)\}\Gamma_s \end{array}\right)\right) \rightarrow \mathcal{I}_{r,s,k,l}$$

The condition $k < l$ ensures that we only capture data flow-dependence and not data anti-dependence.

Example 7.1 *Consider the loop from the array reversal Example 1.1. When computing the effect of the generic loop iteration, we get one success leaf with the following update:*

$$\{\texttt{tmp} := \texttt{a[i]},\ \texttt{a[i]} := \texttt{a[a.length - 1 - i]},\ \texttt{a[a.length - 1 - i]} := \texttt{a[i]}\}$$

The dependence constraint $I_{0,0,k,l}$ is false only if `a.length - 1 -` $\mathit{iter}(l) \doteq \mathit{iter}(k)$ *holds. In the example we have $\mathit{iter}(n) = n$, so this can be simplified to* `a.length - 1` $\doteq k + l$.

In order for $\mathcal{C}$ to be true we need to show that there are no iteration numbers k and l, such that the above equality holds. From the guard specification we obtain that the maximum iteration number is `a.length / 2 - 1`. *The maximum value of $k + l$ is, therefore,* `a.length - 3` *which is not equal to* `a.length - 1`. *This makes $\mathcal{C}$ true and means that the loop does not contain any dependencies that cannot be handled by our method.* □

Computing *mod*$(\vec{\mathcal{U}}, \mathcal{S})$

In Section 4 we used $\mathit{mod}(\vec{\mathcal{U}}, \mathcal{S})$ to compute the set of those locations in $\mathcal{S}$ whose assigned term in $\mathcal{U}$ differs from its assigned term in $\mathcal{S}$. This is very similar to an

Atomic updates

$\texttt{v} := \texttt{val}\ \delta^o\ \texttt{v} := \texttt{val}'$	=	$\{\texttt{v}\}$ **when** $\texttt{val} \not\doteq \texttt{val'}$
$\texttt{o.f} := \texttt{val}\ \delta^o\ \texttt{o'.f} := \texttt{val}'$	=	$\{\texttt{o'.f}\}$ **when** $\texttt{o} \doteq \texttt{o'} \wedge \texttt{val} \not\doteq \texttt{val'}$
$\texttt{a[i]} := \texttt{val}\ \delta^o\ \texttt{b[j]} := \texttt{val}'$	=	$\{\texttt{b[j]}\}$ **when** $\texttt{a} \doteq \texttt{b} \wedge \texttt{i} \doteq \texttt{j} \wedge \texttt{val} \not\doteq \texttt{val'}$
$\_\ \delta^o\ \_$	=	$\emptyset$

General updates

$\mathcal{U}\ \delta^o\ \backslash\texttt{if}\ \ (b)\ \{\mathcal{U}'\}$	=	$\mathcal{U}\ \delta^o\ \mathcal{U}'$ **when** b
$\backslash\texttt{if}\ \ (b)\ \{\mathcal{U}\}\ \delta^o\ \mathcal{U}'$	=	$\mathcal{U}\ \delta^o\ \mathcal{U}'$ **when** b
$\mathcal{U}\ \delta^o\ \backslash\texttt{for}\ T\ s; \mathcal{U}'(s)$	=	$\bigcup_s \mathcal{U}\ \delta^o\ \mathcal{U}'(s)$
$\backslash\texttt{for}\ T\ s; \mathcal{U}(s)\ \delta^o\ \mathcal{U}'$	=	$\bigcup_s \mathcal{U}(s)\ \delta^o\ \mathcal{U}'$
$\mathcal{U}_0, \ldots, \mathcal{U}_m\ \delta^o\ \mathcal{U}'_0, \ldots, \mathcal{U}'_n$	=	$\bigcup_{i,j} \mathcal{U}_i\ \delta^o \mathcal{U}'_j$

Table 2: Computing output dependence constraints.

output dependence analysis. If a location is assigned a different term in $\mathcal{U}$ and $\mathcal{S}$ there will be an output dependence between them. Similarly as above, we define in Table 2 a function δ^o that gives the set of locations, where the terms in its two update arguments differ. The fourth case in the part for atomic updates in Table 2 is the default that is used when none of the other cases applies.

It is sometimes not possible to decide the **when** side conditions in Table 2. In this case we approximate conservatively and assume they are true. Possibly, we remove too much information this way, but the method remains sound. If the second argument is a quantified update, the set of locations could potentially be very large which would make the computation of δ^o very expensive. This can, however, not happen since quantified updates cannot occur in the updates computed for the generic loop iteration.

Another possibility when a side condition cannot be decided would be to compute two different results, one result for when the condition is true and one for when it is false. A problem with this approach is that it potentially doubles the number of returned results each time a side condition cannot be decided. The returned result is used for the computation of the generic loop iteration and, therefore, returning many results would lead to many different generic loop iterations where each needs to be analyzed by the dependence analysis.

Further details on the implementation of the dependence analysis are in [Sch07].

8 Constructing the State Update

If we can show that the iterations of a loop are independent of each other (i.e., the constraint $\mathcal{C}$ defined in the previous section holds), we can capture all state modifications of the loop in one update. Concretely, we use the following quantified update (GR_n, Γ_r, *iter*, and $\mathcal{U}_r$ were defined in Sections 4 and 5):

$$\mathcal{U}_{loop} \equiv \texttt{\textbackslash for int } n; \{\texttt{\textbackslash if } (GR_n) \{\{\texttt{i} := \mathit{iter}(n)\} \bigcup_r \texttt{\textbackslash if } (\Gamma_r) \{\mathcal{U}_r\}\}\} \qquad (4)$$

The innermost conditional update in (4) corresponds to one loop iteration, where the loop variable `i` has the value $\mathit{iter}(n)$. In each state only one Γ_r can be true so we do not need to ensure any particular order of the updates $\vec{\mathcal{U}}$.

The guard GR ensures that the iteration number n is within the loop range. We must take care when using last-win clash-semantics to handle data output-dependence. The iteration with the highest iteration number should have priority over all other iterations. This is ensured by the standard well-order on the JAVA integer type.

9 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 (4) 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 7 is valid. Taken together, this yields:

$$\begin{aligned}\mathcal{D} \equiv (&\bigwedge_{\mathcal{F}\in\vec{\mathcal{F}}} \neg(\exists n.GR_n \wedge \{i := \mathit{iter}(n)\}\mathcal{F})) \wedge \\ &\neg(\exists n.GR_n \wedge \{i := \mathit{iter}(n)\}GF) \wedge GT \wedge \mathcal{C}\end{aligned}$$

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 $\mathcal{U}_{loop}$ (4) 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:

$$\dfrac{\dfrac{\text{If not } \Gamma \Rightarrow \mathcal{D}, \text{ use standard induction}}{\Gamma \Rightarrow \mathcal{U}\langle\texttt{for ... ; ...}\rangle\phi, \mathcal{D}} \qquad \dfrac{\Gamma,\ \mathcal{D} \Rightarrow \mathcal{U}\mathcal{U}_{loop}\langle\ldots\rangle\phi}{\Gamma,\ \mathcal{D} \Rightarrow \mathcal{U}\langle\texttt{for ... ; ...}\rangle\phi}}{\dfrac{\Gamma \Rightarrow \mathcal{U}\langle\texttt{for ... ; ...}\rangle\phi}{\vdots}}\ \mathit{cut}$$

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 the succedent of the left branch can make it easier to close.

	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

Table 3: Parallelizable loops in some representative JAVA CARD programs.

10 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). These programs contain 17 loops. Out of these, our method can be applied to five (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 Table 3.

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.

11 Future Work

The coverage of our verification method can be improved in various ways. One example is the function from the iteration number to the 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 computed from a generic loop iteration. Recent work on automatic termination analysis [CPR06] could be tried in the present setting for proving the termination constraint in Section 5.

We intend to develop general programming guidelines that ensure parallelizability of loops. The current trend towards multi-core processors will result in

more code being written in such a way that it is parallelizable and will for sure rekindle the interest in parallelizability.

Critical dependencies exhibited during dependence analysis are likely to cause complications even in a proof attempt based on a more general proof method such as invariants or induction. Hence, one could try to use the information obtained from the dependence analysis to guide the generalization of, for example, loop invariants.

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

So far our verification method has been worked out and implemented for loop structures, however, it can be seen as a particular instance of a modular approach to proving correctness of non-linear programs composed of code pieces `p(i)` parameterized by some `i`:

1. Compute automatically the effect $\mathcal{U}_{\mathtt{p}}(\mathtt{i})$ of `p(i)` with respect to a given precondition.

2. Using the dependence analysis on $\mathcal{U}_{\mathtt{p}}(\mathtt{i})$, compute a sufficient condition $\mathcal{C}$ under which the code `p(i)` can be seen as *modular* with respect to different iterations of the parameter `i`.

3. The result of the analysis can be used in non-linear composition of `p(i)` as done here for iterative control structures. The idea is just as well applicable for recursive method calls and concurrent processes as is illustrated by the following example:

   ```
   int[] a = new int[n];
   for (int i = 0; i < n; i++) {
     new MyThread(i,a).start();
   }
   ```

   If we assume that the `run()` method of the class `MyThread` updates exactly position `i` of the array `a`, then the effect can be easily captured by an update obtained from executing `run()` in the instance created by `new MyThread(i,a);`. One difference to loops is that in the context of threads one would probably exclude data output-dependence (see Section 6.1) unless assumptions about the scheduler can be made. Otherwise, the inherently parallel structure of state updates is well suited to model concurrent threads.

In this paper we do not discuss in detail what happens after a loop has been transformed into a quantified update. This is outside the scope of the present work. So far, the KeY theorem prover has limited capabilities for automatic reasoning over first-order quantified updates. Since quantified updates occur in many other scenarios [Rüm06] it is worth to spend more effort on that front.

12 Conclusion

We presented a method for formal verification of loops that works by transforming loops into automatable 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. The method can be applied to most initialization and array copy loops but also to more complex loops as witnessed by Example 1.1.

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, Rüm06]. 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.

Acknowledgments

Many thanks to Richard Bubel whose help with the prototypic implementation was invaluable. Max Schröder did the final implementation [Sch07] which is the basis for Section 7. Thanks are also due to Philipp Rümmer for many inspiring discussions. The valuable comments of the anonymous reviewer led to several improvements.

Bibliography

[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.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[BJ88] Robert S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

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

[BRL03] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In *Proc. Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[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 Michael I. Schwartzbach and Thomas Ball, editors, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Ottawa, Canada*, pages 415–426. ACM Press, 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.

[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.

[Jav03] Sun Microsystems, Inc., Santa Clara, California, USA. JAVA CARD *2.2.1 Application Programming Interface*, October 2003.

[JMR04] Bart Jacobs, Claude Marché, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *Proc. 10th Conf. on Algebraic Methodology and Software Technology (AMAST), Stirling, UK*, volume 3116 of *LNCS*, pages 241–257. Springer, July 2004.

[LL05] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In Kwangkeun Yi, editor, *Proc. Progr. Lang. and Systems, 3rd Asian Symposium, Tsukuba*, volume 3780 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 2005.

[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.

[MPM05] Claude Marché and Christine Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *Proc. 18th Intl. Conference on Theorem Proving in Higher Order Logics, Oxford, UK*, volume 3603 of *LNCS*, pages 179–194. Springer, August 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.

[PHM99] Arndt Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.

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

[RCK05] Enric Rodríguez-Carbonell and Deepak Kapur. Program verification using automatic generation of invariants. In Zhiming Liu and Keijiro Araki, editors, *Proc. Theoretical Aspects of Computing (ICTAC),*

Guiyang, China, Revised Selected Papers, volume 3407 of *LNCS*, pages 325–340. Springer-Verlag, 2005.

[Rüm06] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In Miki Hermann and Andrei Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.

[Sch07] Max Schroeder. Using a symbolic dependence analysis for verification of programs containing loops. Master's thesis, Department of Computer Science, University of Karlsruhe, 2007.

[Ste05] Kurt Stenzel. *Verification of Java Card Programs.* PhD thesis, Fakultät für angewandte Informatik, University of Augsburg, 2005.

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

Abstract Interpretation Plugins for Type Systems

Tobias Gedell Daniel Hedin

Abstract

The precision of many type based analyses can be significantly increased given additional information about the programs' execution. For this reason it is not uncommon for such analyses to integrate supporting analyses computing, for instance, nil-pointer or alias information. Such integration is problematic for a number of reasons: 1) it obscures the original intention of the type system especially if multiple additional analyses are added, 2) it makes use of already available analyses difficult, since they have to be rephrased as type systems, and 3) it is non-modular: changing the supporting analyses implies changing the entire type system.

Using ideas from abstract interpretation we present a method for parameterizing type systems over the results of abstract analyses in such a way that one modular correctness proof can be obtained. This is achieved by defining a general format for information transferal and use of the information provided by the abstract analyses. The key gain from this method is a clear separation between the correctness of the analyses and the type system, both in the implementation and correctness proof, which leads to a comparatively easy way of changing the parameterized analysis, and making use of precise, and hence complicated analyses.

In addition, we exemplify the use of the framework by presenting a parameterized type system that uses additional information to improve the precision of exception types in a small imperative language with arrays.

1 Introduction

In the book Types and Programming Languages [Pie02] Pierce defines a type system in the following way: "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute".

Pierce limits his definition to the absence of *certain* program behaviors, since many interesting (bad) behaviors cannot be ruled out statically. Well-known examples of this include division by zero, nil-pointer dereference and class cast errors. The standard solution to this is to lift the semantics and include these errors into the valid results of the execution, often in the form of exceptions, and to have the type system rule out all errors not modeled in the semantics, typically in addition to tracking what errors a program may result in.

For a standard type system this solution is adequate; the types of programs are not affected in any other way than the addition of a set of possible exceptions. In particular, any inaccuracies in the set of possible exceptions are unproblematic to the type system itself (albeit inconvenient to the programmer), and, thus, in standard programming languages not much effort is made to rule out syntactically present but semantically impossible exceptions.

For type based program analyses, however, the situation is different. Not only are we not able to change the semantics to fit the capabilities of the type system, since we are, in effect, retrofitting a type system onto an existing language, but for some analyses — notably, for *information flow security* type systems [SM03] — inaccuracies propagate from e.g. the exception types via *implicit flows* to the other types lowering the precision of the type system and possibly rendering more semantically secure programs to be classified as insecure. Consider the following example:

```
try c1; c2; ...  catch (Exception e) ch
```

If the command c_1 may fail this affects whether the succeeding commands $c_2, \ldots$ are run or not, and thus any side effects — e.g. output on a public network — will encode information about the data manipulated by c_1. If this information must be protected, this put serious limits on the succeeding commands $c_2, \ldots$ and on the exception handler *ch*.

This is problematic, since dynamic error handling introduces many possible branches — every partial instruction becomes a possible branch to the error handler *if it cannot be guaranteed not to crash*, and, thus, a source of implicit flows. Hence, from a practical standpoint, there is a need to increase the accuracy of type based information flow analyses as demonstrated by some recent attempts [BPR07, HS06, ABB06]. Noting that the majority of the information flow analyses are formulated in terms of type systems, we focus on how to strengthen a type system with additional information to increase its accuracy.

Even though our main motivation for this work comes from information flow type systems, we investigate the problem in terms of a standard type system; this both generalizes the method and simplifies the presentation. All our results are immediately applicable to information flow type systems.

We see two major different methods of solving the problem of strengthening type systems: 1) by *integration*, and 2) by *parameterization*. Briefly, 1) relies on extending the type system to compute the additional needed information, and 2) relies on using information about the programs' execution provided by other analyses. Integration is problematic for a number of reasons: 1) it obscures the original intention of the type system especially if multiple additional analyses are added, 2) it makes use of already available analyses difficult, since they have to be rephrased as type systems, and 3) it is non-modular: changing the supporting analysis implies changing the entire type system.

Contribution We present a modular approach for parameterizing type systems with information about the program execution; the method is modular not only at the type system level, but also at the proof level.

The novelty of the approach lies not in the idea of parameterizing information in itself, rather, the novelty is the setting — the parameterization of *type systems* with information from *abstract analyses* — together with the identification of a general, widely applicable format for information passing and inspection, which allows for modularity with only small modifications to the type system and its correctness proof, and no modifications to the abstract analyses. This modularity makes instantiating parameterized type systems with the results of different abstract analyses relatively cheap, which can be leveraged to create staged type systems, where increasingly precise analyses are chosen based on previous typing errors.

Finally, we exemplify the use of the method in terms of a parameterized type system for a small imperative language with dynamically allocated arrays, and explore how the parameterization can be used to rule out nil-pointer exceptions, and exceptions stemming from array indices outside the bounds of the corresponding arrays.

Outline Section 2 presents a small imperative language with arrays, used to explain the method more concretely. Section 3 presents the parameterization: the abstract environment maps, the plugins properties and the plugins, and describes the process of parameterizing a type system. Section 4 is a concrete example of applying the method to get a parameterized type system of the language of Section 2. Section 5 discusses related work, and finally Section 6 concludes and discusses future work.

2 Language

To be concrete we use a small imperative language with arrays to illustrate our method.

Syntax The language is a standard while language with arrays. For simplicity we consider all binary operators to be total; the same techniques described to handling the partiality of array indexing apply to partial operators. The syntax of the language is found in Figure 1, where the allocation type $\tau[i]$ indicates that an array of size i with elements of type τ should be allocated; the primitive types ranged over by τ are defined in Section 4.1 below.

$$\begin{array}{lrll} \text{Expressions} & e & ::= & \mathit{nil} \mid i \mid x \mid e_1 \star e_2 \mid x[e] \mid \mathit{len}(x) \\ \text{Commands} & c & ::= & x := e \mid x_1[e] := x_2 \mid \mathit{if}\ e\ c_1\ c_2 \mid \mathit{while}\ e\ c \mid \\ & & & c_1; c_2 \mid x := \mathit{new}(\tau[i]) \mid \mathit{skip} \end{array}$$

Figure 1: Syntax

Semantics The semantics of the expressions found in Figure 2 is given in terms of a big step semantics with transitions of the form $\langle E, e\rangle \Downarrow v_\perp$, where $v_\perp$ ranges over error lifted values, i.e., the set of values extended with $\perp$, which indicates errors, and E ranges over the set of environments Env, i.e., pairs (s, h) of stores and heaps. The values consist of the integers i and the pointers p. The arrays a are pairs (i, d) of the size of the array and a map from integers to values with a contiguous domain starting from 0. Formally, d ranges over $\bigcup_{n\in\mathbb{N}}\{[0 \mapsto v_1, \ldots, n \mapsto v_n]\}$. The stores s are maps from variables x to values, and the heaps h are maps from pointers to arrays. In the definition of the

$$\frac{}{\langle E, nil\rangle \Downarrow nil} \qquad \frac{}{\langle E, i\rangle \Downarrow i} \qquad \frac{E(x) = v}{\langle E, x\rangle \Downarrow v}$$

$$\frac{E(x) = nil}{\langle E, x[e]\rangle \Downarrow \perp} \qquad \frac{E(x) = p \quad E(p) = (i, d)}{\langle E, len(x)\rangle \Downarrow i} \qquad \frac{E(x) = nil}{\langle E, len(x)\rangle \Downarrow \perp}$$

$$\frac{E(x) = p \quad E(p) = (i, d) \quad \langle E, e\rangle \Downarrow \perp}{\langle E, x[e]\rangle \Downarrow \perp}$$

$$\frac{E(x) = p \quad E(p) = (i_1, d) \quad \langle E, e\rangle \Downarrow i_2 \quad i_2 \notin [0..i_1 - 1]}{\langle E, x[e]\rangle \Downarrow \perp}$$

$$\frac{E(x) = p \quad E(p) = (i_1, d) \quad \langle E, e\rangle \Downarrow i_2 \quad i_2 \in [0..i_1 - 1]}{\langle E, x[e]\rangle \Downarrow d(i_2)}$$

$$\frac{\langle E, e_1\rangle \Downarrow \perp}{\langle E, e_1 \star e_2\rangle \Downarrow \perp} \qquad \frac{\langle E, e_1\rangle \Downarrow v_1 \quad \langle E, e_2\rangle \Downarrow \perp}{\langle E, e_1 \star e_2\rangle \Downarrow \perp}$$

$$\frac{\langle E, e_1\rangle \Downarrow v_1 \quad \langle E, e_2\rangle \Downarrow v_2}{\langle E, e_1 \star e_2\rangle \Downarrow v_1 \star v_2}$$

Figure 2: Semantic Rules of Expressions

semantics, if $a = (i_1, d)$ then let $a(i_2)$ denote $d(i_2)$. Further, for $E = (s, h)$, let $E(x)$ denote $s(x)$, $E[x \mapsto v]$ denote $(s[x \mapsto v], h)$, $E(p)$ denote $h(p)$, and similarly for other operations on environments including variables or pointers.

The semantics of commands found in Figure 3 is given in terms of a small step semantics between configurations C with transitions of the form $\langle E, c\rangle \to C$, where C is either one of the terminal configurations $\perp_E$ and $\langle E, \mathit{skip}\rangle$ indicating abnormal and normal termination in the environment E, respectively, or a non-terminal configuration $\langle E, c\rangle$ where $c \neq \mathit{skip}$.

As is common for small step semantics we use evaluation contexts R.

$$R ::= \cdot \mid x := R \mid x[R] := x \mid \mathit{if}\ R\ c\ c \mid R; c$$

The accompanying standard reduction rules found in Figure 4 allow for leftmost reduction of sequences, error propagation and reduction of expressions inside commands.

$$\frac{}{\langle E, x := v\rangle \to \langle E[x \mapsto v], \mathit{skip}\rangle} \qquad \frac{E(x_1) = \mathit{nil}}{\langle E, x_1[i] := x_2\rangle \to \bot_E}$$

$$\frac{E(x_1) = p \quad E(p) = (i_2, d) \quad E(x_2) = v \quad i_1 \notin [0..(i_2 - 1)]}{\langle E, x_1[i_1] := x_2\rangle \to \bot_E}$$

$$\frac{E(x_1) = p \quad E(p) = (i_2, d) \quad E(x_2) = v \quad i_1 \in [0..(i_2 - 1)]}{\langle E, x_1[i_1] := x_2\rangle \to \langle E[p \mapsto (i_2, d[i_1 \mapsto v])], \mathit{skip}\rangle}$$

$$\frac{}{\langle E, \mathit{while}\ e\ c\rangle \to \langle E, \mathit{if}\ e\ (c; \mathit{while}\ e\ c)\ \mathit{skip}\rangle}$$

$$\frac{v \neq 0}{\langle E, \mathit{if}\ v\ c_1\ c_2\rangle \to \langle E, c_1\rangle} \qquad \frac{}{\langle E, \mathit{if}\ 0\ c_1\ c_2\rangle \to \langle E, c_2\rangle}$$

$$\frac{}{\langle E, \mathit{skip}; c\rangle \to \langle E, c\rangle}$$

$$\frac{a = \mathit{array}(\tau, i) \quad p \notin \mathit{dom}(h)}{\langle (s, h), x := \mathit{new}(\tau[i])\rangle \to \langle (s[x \mapsto p], h[p \mapsto a]), \mathit{skip}\rangle}$$

Figure 3: Semantic Rules for Commands

$$\frac{\langle E, e\rangle \Downarrow v}{\langle E, R[e]\rangle \to \langle E, R[v]\rangle} \qquad \frac{\langle E, e\rangle \Downarrow \bot}{\langle E, R[e]\rangle \to \bot_E}$$

$$\frac{\langle E_1, c_1\rangle \to \langle E_2, c_2\rangle}{\langle E_1, R[c_1]\rangle \to \langle E_2, R[c_2]\rangle} \qquad \frac{\langle E, c\rangle \to \bot_E}{\langle E, R[c]\rangle \to \bot_E}$$

Figure 4: Semantic Rules for Contexts

3 Parameterization

With this we are ready to detail the method of parameterization. First, let us recapture our goal: we want to describe a modular way of parameterizing a type system with information about the programs' execution in such a way that a modular correctness proof can be formed for the resulting system, with the property that an instantiated system satisfies a correspondingly instantiated correctness proof.

To achieve this, we define a general format of parameterized information and a general method to access this information. Using the ideas of abstract interpretation, we let the parameterized information be a map from program points to abstract environments, intuitively representing the set of environments that can reach each program point. Such a map is semantically sound — a *solution* in our terminology — with respect to a set of initial concrete environments and a program, if every possible execution trace the initial environments can give rise to is modeled by the map.

For modularity we do not want to assume anything about the structure of the abstract environments, but treat them as completely opaque. Noting that each type system uses a finite number of forms of questions, we parameterize

the type system not only over the abstract environment map, but also over a set of *plugins* — sound approximations of the semantic properties of the questions used by the type system.

Labeled Commands Following the elegant approach of Sands and Hunt [HS08] we extend the command language with label annotations, which allow for a particularly direct way of recording the environments that enter and leave the labeled commands. Let l range over labels drawn from the set of labels $\mathcal{L}$. A command c can be annotated with an entry label $(c)^l$, an exit label $(c)_l$, or both.

For while loops, care must be taken so that the entry label records all environments that may flow into the loop, including the environments produced by the iterations of the loop. This is important, since the entry label of the loop may be used when typing its guard expression. Thus, the semantic rule for the while loop is changed to pass on the entry label as follows.

$$\frac{}{\langle E, (\mathit{while}\ e\ c)^l\rangle \rightarrow \langle E, (\mathit{if}\ e\ (c; (\mathit{while}\ e\ c)^l)\ \mathit{skip})^l\rangle}$$

For the remaining commands, we extend the reduction contexts with $(R)_l$, which allows for reduction under exit labels, and the semantics with the following transitions.

$$\frac{\langle E_1, c_1\rangle \rightarrow \langle E_2, c_2\rangle}{\langle E_1, (c_1)^l\rangle \rightarrow \langle E_2, c_2\rangle} \qquad \frac{}{\langle E, (\mathit{skip})_l\rangle \rightarrow \langle E, \mathit{skip}\rangle}$$

The idea is that a transition of the form $\langle E_1, (c_1)^l\rangle \rightarrow \langle E_2, c_2\rangle$ leaves a marker in the execution sequence that the command labeled with the entry label l was executed in E, and a transition of the form $\langle E, (\mathit{skip})_l\rangle \rightarrow \langle E, \mathit{skip}\rangle$ indicates that the environment E was produced by the command labeled with the exit label l, which is why allowing reduction under exit labels but not under entry labels is important.

3.1 Abstract Environment Maps

Using the ideas of abstract interpretation [CC77, CC79], let $\mathbb{Env}$ be the set of abstract environments ranged over by $\mathbb{E}$, equipped with a concretization function $\gamma : \mathbb{Env} \rightarrow \mathcal{P}(\mathit{Env})$, and let an abstract environment map $\mathbb{M} : \mathcal{L} \rightarrow \mathbb{Env}$ be a map from program points to abstract environments, associating each program point with an abstract environment representing all concrete environments that may reach the program point.

We define two soundness properties for abstract environment maps that relate the maps to the execution of a program when started in environments drawn from a set of initial environments $\mathcal{C}$.

An abstract environment map $\mathbb{M}$ is an *entry solution* written $\mathit{entrysol}^{E_1}_{c_1}(\mathbb{M})$ with respect to an initial concrete environment E_1, and a program c_1 if all $\langle E_2, (c_2)^l\rangle \rightarrow \langle E_3, c_3\rangle$ transitions in the trace originating in $\langle E_1, c_1\rangle$ are captured

by $\mathbb{M}$. The notion of *exit solution* written $exitsol_c^{E_1}(\mathbb{M})$ is defined similarly but with respect to all transitions of the form $\langle E_2, (skip)_l\rangle \rightarrow \langle E_2, skip\rangle$. We include the notion of exit solutions for completeness, even though they are not used by any of the examples in this paper.

$$\begin{array}{lcl} entrysol_{c_1}^{E_1}(\mathbb{M}) & \equiv & \forall E_2, c_2, l \;.\; \langle E_1, c_1\rangle \rightarrow^* \langle E_2, R[(c_2)^l]\rangle \Longrightarrow E_2 \in \gamma(\mathbb{M}(l)) \\ exitsol_c^{E_1}(\mathbb{M}) & \equiv & \forall E_2, l \;.\; \langle E_1, c\rangle \rightarrow^* \langle E_2, R[(skip)_l]\rangle \Longrightarrow E_2 \in \gamma(\mathbb{M}(l)) \end{array}$$

The definitions are lifted to sets of initial environments $\mathcal{C}$ in the obvious way.

Both the entry and exit solution properties are preserved under execution.

Lemma 3.1 (Preservation of Entry and Exit Solutions under Execution) *In the following, let $\mathcal{C}_1$ be the set of initial concrete environments and $\mathcal{C}_2$ the set of environments that reach c_2, i.e., $\mathcal{C}_2 = \{E_2 \mid E_1 \in \mathcal{C}_1,\ \langle E_1, c_1\rangle \rightarrow \langle E_2, c_2\rangle\}$.*

$$entrysol_{c_1}^{\mathcal{C}_1}(\mathbb{M}) \Longrightarrow entrysol_{c_2}^{\mathcal{C}_2}(\mathbb{M}) \text{ and } exitsol_{c_1}^{\mathcal{C}_1}(\mathbb{M}) \Longrightarrow exitsol_{c_2}^{\mathcal{C}_2}(\mathbb{M})$$

Proof 3.1

Entry Solutions *Assume* (1) $entrysol_{c_1}^{\mathcal{C}_1}(\mathbb{M})$*, i.e., that for all* $E_1 \in \mathcal{C}_1, E_3, c_3, l$ *it holds that* $\langle E_1, c_1\rangle \rightarrow^* \langle E_3, R[(c_3)^l]\rangle \Longrightarrow E_3 \in \gamma(\mathbb{M}(l))$*. We must show* $entrysol_{c_2}^{\mathcal{C}_2}(\mathbb{M})$*, i.e., that* $\forall E_2 \in \mathcal{C}_2, E_3, c_3 \;.\; \langle E_2, c_2\rangle \rightarrow^* \langle E_3, R[(c_3)^l]\rangle \Longrightarrow E_3 \in \gamma(\mathbb{M}(l))$*. Thus, assume* (2) $E_2 \in \mathcal{C}_2$ *and* (3) $\langle E_2, c_2\rangle \rightarrow^* \langle E_3, R[(c_3)^l]\rangle$ *and show that* $E_3 \in \gamma(\mathbb{M}(l))$*. From* (2) *and* (3) *we get* $\langle E_1, c_1\rangle \rightarrow^* \langle E_3, R[(c_3)^l]\rangle$ *for some* $E_1 \in \mathcal{C}_1$*. This, together with* (1) *gives* $E_3 \in \gamma(\mathbb{M}(l))$ *and we are done.*

Exit Solutions *The proof of preservation for exit solutions is identical to the proof for entry solutions, since the properties are of the same form.*

It should be pointed out that solutions can freely be paired to form new solutions similarly to the independent attribute method for abstract interpretation [CC79]. This is important since it shows that no generality is lost by parameterizing a type system over only one abstract environment map.

3.2 Plugins

To the parameterized type systems, the structure of the abstract environments is opaque and cannot be accessed directly. This allows for the decoupling of the parameterized type system and the external analysis computing the abstract environments. However, the parameterized type systems need a way to extract the desired information. To this end we introduce the concept of *plugins*. Intuitively, a plugin provides information about a specific property of an environment; for instance, a nil-pointer plugin provides information about which parts of the environment are nil.

The plugins are defined to be sound approximations of *plugin properties*, defined as families of relations on expressions.

Plugin Properties Let R be an n-ary relation on values; R induces a *plugin property*, written $R^\diamond$, which is a family of n-ary relations on expressions indexed by environments defined as follows.

$$(e_1, \ldots, e_n) \in R^\diamond_E \iff \langle E, e_1\rangle \Downarrow v_1 \wedge \ldots \wedge \langle E, e_n\rangle \Downarrow v_n \Longrightarrow (v_1, \ldots, v_n) \in R$$

We can use the expression language to define semantic properties about environments, since the expression language is simple, in particular, since it does not contain iteration, and is free from side effects. A major advantage of the approach is that it allows for a relatively simple treatment of expressions in programs.

The choice of using the expression language as the plugin language is merely out of convenience — languages with richer expression language would mandate a separate language for the plugins and treat the exceptions similarly to the statement, i.e., extend the labeling and the solutions to the expressions. In our case, however, a separate plugin language would be identical to the expressions.

Example 3.1 (Non-nil and Less-than Plugin Properties) *The non-nil plugin property $nn^\diamond$ can be defined by a family of predicates indexed over concrete environments induced by the value property nn defined such that $nn(v)$ holds only if the value v is not equal to nil. Similarly, the less-than plugin property $lt^\diamond$ can be defined by lt such that $lt(v_1, v_2)$ holds only if the value v_1 is less than the value v_2.* □

Plugins A plugin is a family of relations on expressions indexed by abstract environments. Given a plugin property we define the *corresponding plugins* to be relations on expressions indexed by abstract environments satisfying the following demand. Let $R^\sharp$ denote a plugin corresponding to the plugin property $R^\diamond$.

$$(e_1, \ldots, e_n) \in R^\sharp_{\mathbb{E}} \Longrightarrow \forall E \in \gamma(\mathbb{E}).\ (e_1, \ldots, e_n) \in R^\diamond_E$$

It is important to note that for each plugin property there are many possible plugins, since the above formulation allows for approximative plugins. This means that regardless of the abstract environment, and the decidability of the plugin property $R^\diamond$, there exist decidable plugins, which guarantees the possibility of preservation of decidability for parameterized type systems.

Example 3.2 (Use of Plugins) *Assume a type system computing a set of possibly thrown exceptions. When typing, for example, the array length operator $len(x)$ we are interested in the plugin property given by the non-nil predicate nn. Let $\mathbb{E}$ be a sound representation of all environments reaching $len(x)$. Given $nn^\sharp_{\mathbb{E}}(x)$, we know that x will not be nil in any of the concrete environments represented by $\mathbb{E}$, and, since $\mathbb{E}$ is a sound representation of all environments that can reach the array length operator, we know that a nil-pointer exception will not be thrown.* □

Despite the relative simplicity of the plugin format it is surprisingly powerful; in addition to the obvious information, such as is x ever nil, it turns out that plugins can be used to explore the structure of the heap as we show in [GH08].

3.3 Overview of a Parameterized Type System

Assume an arbitrary flow insensitive type system — the method works equally well for flow sensitive type systems, but for brevity of explanation this section is done in terms of a flow insensitive system — of the form $\Gamma \vdash_A c$ expressing that c is well-typed in the type signature Γ, under the additional assumption A. We let the exact forms of Γ and A be abstract; however, typical examples are that Γ is a store type, and, for information flow type systems, that A is the security level of the context, known as the pc [SM03]. Also local to this section, assume a big-step semantics of the form $\langle E_1, c\rangle \rightarrow E_2$, read c executes in the environment E_1 resulting in the environment E_2.

The first step in parameterizing the type system is to identify the plugin properties $R_1^\diamond, \ldots, R_m^\diamond$ that are to be used in the parameterized type rules. For instance the non-nil plugin property can be used to increase the precision of the type rule for the array length operator as discussed in Example 3.2 above, cf. the corresponding type rules in Section 4 below. Each type rule is then parameterized with an abstract environment map $\mathbb{M}$, and a number of plugins $R_1^\sharp, \ldots R_m^\sharp$, one for each of the plugin properties, forming a parameterized system of the following form.

$$\Gamma \vdash_A^{\mathbb{M}, R_1^\sharp, \ldots, R_m^\sharp} c$$

A typical correctness argument for type systems is *preservation* [Pie02], i.e., the preservation of a type induced invariant, well-formedness, (see Section 4.3 below) under execution. Well-formedness defines when an environment conforms to an environment type, e.g. that all variables of integer type contain integers. Let $wf_\Gamma(E)$ denote that E is well-formed in the environment type Γ; a typical preservation statement has the following form:

$$\Gamma \vdash_A c \Longrightarrow wf_\Gamma(E_1) \wedge \langle E_1, c\rangle \rightarrow E_2 \Longrightarrow wf_\Gamma(E_2)$$

More generally, a class of correctness arguments for type systems have the form of preservation of an arbitrary type indexed relation $\mathcal{R}_\Gamma$ under multiple executions:

$$\begin{aligned}\Gamma \vdash_A c \Longrightarrow \mathcal{R}_\Gamma(E_{11}, \ldots, E_{1n}) \wedge \\ \langle E_{11}, c\rangle \rightarrow E_{21} \wedge \cdots \wedge \langle E_{1n}, c\rangle \rightarrow E_{2n} \Longrightarrow \mathcal{R}_\Gamma(E_{21}, \ldots, E_{2n})\end{aligned}$$

This generalization is needed to capture invariants that are not *safety properties*, for instance noninterference or live variable analysis.

For *conservative* parameterizations, i.e., where we add type rules with increased precision, the proofs of correctness are essentially identical to the old proofs, where certain execution cases have been ruled out using the semantic interpretation of the plugins. To see this consider that a typical proof of the above lemma proceeds with a case analysis on the possible ways c can execute in the different environments E_{11} to E_{1n} and proves the property for each case. See the proof of Theorem 4.1 in Section 4.3 for an example of this. The correctness

statement for the parameterized types system becomes:

$$entrysol_c^{\mathcal{C}}(\mathbb{M}) \wedge E_{11} \in \mathcal{C} \wedge \cdots \wedge E_{1n} \in \mathcal{C} \wedge \Gamma \vdash_A^{\mathbb{M},R_1^\sharp,\ldots,R_m^\sharp} c \Longrightarrow$$
$$\mathcal{R}_\Gamma(E_{11},\ldots,E_{1n}) \wedge \langle E_{11}, c\rangle \rightarrow E_{21} \wedge \cdots \wedge \langle E_{1n}, c\rangle \rightarrow E_{2n} \Longrightarrow \mathcal{R}_\Gamma(E_{21},\ldots,E_{2n})$$

The interpretation of this statement is that execution started in any of the environments in the set of possible initial environments is $\mathcal{R}_\Gamma$-preserving, i.e., it narrows the validity of the original lemma to the set of initial environments.

Proof of Correctness

We have shown a way to define a type system relative to one or more abstract environment maps, and a number of plugins and how to prove its conditional correctness, i.e., by assuming that the abstract environment maps are solutions, and that the plugins approximate the corresponding plugin properties.

To instantiate the type system with the result of an external analysis, it must first be established that the results of the analysis are correct according to our notion of solutions. This proof can typically be done once per family of static analyses.

What is left per instantiation is to show that the used plugins are correct, i.e., that they are sound approximations of the corresponding plugin properties with respect to the semantics of the abstract environments. In most cases, this is trivial since the structure of the abstract environments have been chosen with this in mind. Furthermore, many type systems can be improved with similar information; thus, it should be possible to build a library with plugins for different plugin properties that can be used when instantiating implementations of parameterized type systems.

The conclusion is that creating new correct instantiations can be made a relatively cheap operation, which leads to interesting implementation possibilities.

3.4 Instantiations and Staged Type Systems

With this we are ready to investigate possible implementations of a parameterized type system. For each program c, and type signature Γ, A we get a set of possible derivations of $\Gamma \vdash_A c$. If this set is inhabited we say that c is type correct with respect to the type signature Γ, A, otherwise it is type incorrect. The type checking problem for a standard type system is to decide, for a given command and type signature, whether this set of derivations is empty or not.

For a parameterized type system we get, for each program c, type signature Γ, A, and set of possible initial environments $\mathcal{C}$, a set of possible derivations of $\Gamma \vdash_A^{\mathbb{M},R_1^\sharp,\ldots,R_n^\sharp} c$ for each solution $\mathbb{M}$, and possible plugins $R_1^\sharp, \ldots, R_n^\sharp$. These sets can be illustrated in a matrix with the solutions as columns and all combinations of the plugins as the rows. To limit the size of the matrix we only consider two

plugin properties, $R_1^\diamond$ and $R_2^\diamond$.

$\Gamma, A, c, \mathcal{C}$	$\mathbb{M}_1$	$\mathbb{M}_2$	$\ldots$	$\mathbb{M}_n$
$R_{11}^\sharp, R_{21}^\sharp$	T_{11}^1	T_{11}^2	$\ldots$	T_{11}^n
$R_{11}^\sharp, R_{22}^\sharp$	T_{12}^1	T_{12}^2	$\ldots$	T_{12}^n
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$R_{1m}^\sharp, R_{2p}^\sharp$	T_{mp}^1	T_{mp}^2	$\ldots$	T_{mp}^n

$\mathbb{M}$ ranges over the different solutions, $R_i^\sharp$ ranges over different plugins for the plugin property $R_i^\diamond$, and each T_{kl}^i is the set of possible derivations of $\Gamma \vdash_A^{\mathbb{M}_i, R_{1k}^\sharp, R_{2l}^\sharp} c$ where the plugins are instantiated with the concretization function of $\mathbb{M}_i$. The type checking problem for a parameterized type system becomes deciding whether at least one of the derivation sets T_{kl}^i is non-empty in which case c is type correct with respect to the type signature Γ, A and the set of possible initial environments $\mathcal{C}$.

Implementation of Parameterized Type Systems

An implementation of a unparameterized type system is a function tc that takes a type signature and a program such that $tc(\Gamma, A, c)$ returns *true* if it can find a derivation of $\Gamma \vdash_A c$.

For simplicity, assume a fixed set of abstract environments $\mathbb{Env}$, a number of corresponding abstract analyses AI_i, and a number of corresponding decision procedures d_j for the plugin properties $R_j^\diamond$. An implementation of a parameterized type system is a function tc such that $tc(\Gamma, A, c, \mathbb{M}, d_1, \ldots, d_n)$ returns *true* if there is a derivation of $\Gamma \vdash_A^{\mathbb{M}, R_1^\sharp, \ldots, R_n^\sharp} c$ where $R_i^\sharp$ is the extension of d_i.

The decision procedures and the results of the abstract analyses correspond to selecting a number of rows and columns respectively in the above matrix. The resulting type checking problem can be illustrated as follows, where f is defined by $f(\mathbb{M}, d_1, d_2) \equiv tc(\Gamma, A, c, \mathbb{M}, d_1, d_2)$.

$\Gamma, A, c, \mathcal{C}, \mathbb{Env}$	$AI_1 \to \mathbb{M}_1$	$AI_2 \to \mathbb{M}_2$	$\ldots$	$AI_n \to \mathbb{M}_n$
d_{11}, d_{21}	$f(\mathbb{M}_1, d_{11}, d_{21})$	$f(\mathbb{M}_2, d_{11}, d_{21})$	$\ldots$	$f(\mathbb{M}_n, d_{11}, d_{21})$
d_{11}, d_{22}	$f(\mathbb{M}_1, d_{11}, d_{22})$	$f(\mathbb{M}_2, d_{11}, d_{22})$	$\ldots$	$f(\mathbb{M}_n, d_{11}, d_{22})$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
d_{1m}, d_{2p}	$f(\mathbb{M}_1, d_{1m}, d_{2p})$	$f(\mathbb{M}_2, d_{1m}, d_{2p})$	$\ldots$	$f(\mathbb{M}_n, d_{1m}, d_{2p})$

Hence, the type checking problem is reduced to finding an abstract analysis AI_i and decision procedures $d_{1k}, d_{2l}, \ldots$ such that $f(\mathbb{M}_i, d_{1k}, d_{2l}, \ldots)$ is non-empty.

Staged Type Systems

The resulting type checking problem is a search problem and the search order will clearly have a major impact on the efficiency of the implementation. One possible search strategy is to order the abstract analyses and decision procedures

according to complexity and runtime cost and form a *staged* type system, that successively invokes more and more expensive and precise analyses and decision procedures.

For instance, assume a decision procedure f for a parameterized type system, defined as above. Also assume a number of external analyses AI_1 to AI_n of increasing complexity and precision chosen from a family of analyses that guarantees valid environment maps. Assuming that the result of these analyses are relevant to the sought after information, i.e., that we for each analysis AI_i can form non-empty sound decision procedures $d_1, \ldots, d_m$ for the results of the analyses. Furthermore, assume a constant external analysis AI_0, whose constant result is a solution, $\mathbb{M}_0$, mapping all labels to the top abstract environment representing all concrete environments, with the empty decision procedure d^0.

With this we can fairly easily implement a staged type system. First, we try to type the program under consideration with respect to $f(\mathbb{M}_0, d^0, \ldots, d^0)$ using the environment map extracted from the constant analysis and the empty decision procedure for all plugin properties. Given that the type system has been conservatively parameterized this is equivalent to checking with the original, non-parameterized, type system. If the check fails, the failure may come from lack of precision, and we may try with more precise external analyses. Thus, for every failure at stage i, the system uses AI_{i+1} to compute an environment map $\mathbb{M}_{i+1}$ and tries to type check using $f(\mathbb{M}_{i+1}, d_1, \ldots, d_m)$.

More intricate staged type systems can also be formed where the reason for the type failure is analyzed and given as feedback to the next stage. The benefit of doing this is apparent in cases where the environment map is a combination of the result of a number of external analysis. Assume for example a parameterized type system using both aliasing information and integer domain information. If the program fails to type in the first stage because of an alias problem, it would probably suffice to only recompute the alias information using a more precise alias analysis in the next stage. One way to view the set of increasingly precise external analysis is as a matrix with one dimension for each type of analysis and plugin property. In the general setting where a parameterized type system uses multiple external analysis the external analysis build up a multi-dimensional matrix where each point corresponds to a particular instantiation of the type system. The type error is then used to navigate through this matrix.

4 A Parameterized Type System

In this section we exemplify the ideas described in the previous section by presenting a parameterized type system for the language introduced in Section 2. The type system improves over the typical type system for such a language by using the parameterized information to rule out exceptions that cannot occur.

A larger example of a parameterized type system, showing how plugins can be used to perform structural weakening and strong updates for a flow-sensitive type system, can be found in [GH08].

4.1 Type Language

The primitive types ranged over by τ are the type of integers *int*, and array types, $\tau[]$, indicating an array with elements of type τ. The store types ranged over by Σ are maps from variables to primitive types. The exception types ranged over by ξ are $\bot_\Sigma$, indicating the possibility that an exception is thrown, and $\top$, indicating that no exception is thrown. This is a simplification from typical models of exceptions, where multiple types are used to indicate the reason for the exception. However, for the purpose of exemplifying the parameterization this model suffices; the results are easily extended to a richer model. In addition we use a standard subtype relation $<:$ defined to be the smallest reflexive, transitive relation satisfying:

$$\frac{}{\top <: \bot_\Sigma} \qquad \frac{\Sigma_1 <: \Sigma_2}{\bot_{\Sigma_1} <: \bot_{\Sigma_2}} \qquad \frac{\forall x \in dom(\Sigma_2)\ .\ \Sigma_1(x) <: \Sigma_2(x)}{\Sigma_1 <: \Sigma_2}$$

4.2 Type Rules

The judgment for expressions, $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e : \tau, \xi$, is read as the expression e is well-typed with respect to the abstract environment $\mathbb{E}$, the non-nil plugin $nn^\sharp$, and the less-than plugin $lt^\sharp$, in the environment type Σ, with return type τ possibly resulting in exceptions as indicated by ξ. The type rules for expressions are found in Figure 5 where $\vdash^\ddagger$ is used as short notation for $\vdash^{\mathbb{E},nn^\sharp,lt^\sharp}$.

$$\frac{}{\Sigma \vdash^\ddagger nil : A, \top} \qquad \frac{}{\Sigma \vdash^\ddagger i : int, \top} \qquad \frac{\Sigma(x) = \tau}{\Sigma \vdash^\ddagger x : \tau, \top}$$

$$\frac{\Sigma \vdash^\ddagger e_1 : \tau, \xi \qquad \Sigma \vdash^\ddagger e_2 : \tau, \xi}{\Sigma \vdash^\ddagger e_1 \star e_2 : \tau, \xi}$$

$$\frac{\Sigma \vdash^\ddagger e : \tau_1, \xi_1 \qquad \tau_1 <: \tau_2 \qquad \xi_1 <: \xi_2}{\Sigma \vdash^\ddagger e : \tau_2, \xi_2}$$

$$\frac{\Sigma(x) = \tau[] \qquad \neg nn^\sharp_{\mathbb{E}}(x)}{\Sigma \vdash^\ddagger len(x) : int, \bot_\Sigma} \qquad \frac{\Sigma(x) = \tau[] \qquad nn^\sharp_{\mathbb{E}}(x)}{\Sigma \vdash^\ddagger len(x) : int, \top}$$

$$\frac{\Sigma(x) = \tau[] \qquad \Sigma \vdash^\ddagger e : int, \xi \qquad nn^\sharp_{\mathbb{E}}(x) \wedge -1\ lt^\sharp_{\mathbb{E}}\ e \wedge e\ lt^\sharp_{\mathbb{E}}\ len(x)}{\Sigma \vdash^\ddagger x[e] : \tau, \xi}$$

$$\frac{\Sigma(x) = \tau[] \qquad \Sigma \vdash^\ddagger e : int, \xi \qquad \neg(nn^\sharp_{\mathbb{E}}(x) \wedge -1\ lt^\sharp_{\mathbb{E}}\ e \wedge e\ lt^\sharp_{\mathbb{E}}\ len(x)) \qquad \bot_\Sigma <: \xi}{\Sigma \vdash^\ddagger x[e] : \tau, \xi}$$

Figure 5: Type Rules for Expressions

The type system for commands is flow-sensitive; the judgment, $\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c \Rightarrow \Sigma_2, \xi$ is read as the command c is well-typed with respect to the abstract

environment map $\mathbb{M}$, the non-nil plugin $nn^\sharp$, and the less-than plugin $lt^\sharp$, in the environment type Σ_1 resulting in the environment type Σ_2, possibly resulting in an exception as indicated by ξ. The type rules for commands are found in Figure 6, where $\vdash^\dagger$ is used as short notation for $\vdash^{\mathbb{M},nn^\sharp,lt^\sharp}$, and $\vdash^\ddagger$ is used as short notation for $\vdash^{\mathbb{M}(l),nn^\sharp,lt^\sharp}$.

$$\frac{\Sigma \vdash^\ddagger e : \tau, \xi}{\Sigma \vdash^\dagger (x := e)^l \Rightarrow \Sigma[x \mapsto \tau], \xi}$$

$$\frac{\begin{array}{c}\Sigma \vdash^\ddagger e : int, \xi \quad \Sigma(x_1) = \tau_1[] \quad \Sigma(x_2) = \tau_2 \quad \tau_2 <: \tau_1 \\ \neg(nn^\sharp_{\mathbb{M}(l)}(x_1) \wedge -1\ lt^\sharp_{\mathbb{M}(l)}\ e \wedge e\ lt^\sharp_{\mathbb{M}(l)}\ len(x_1))\end{array}}{\Sigma \vdash^\dagger (x_1[e] := x_2)^l \Rightarrow \Sigma, \bot_\Sigma}$$

$$\frac{\begin{array}{c}\Sigma \vdash^\ddagger e : int, \xi \quad \Sigma(x_1) = \tau_1[] \quad \Sigma(x_2) = \tau_2 \quad \tau_2 <: \tau_1 \\ nn^\sharp_{\mathbb{M}(l)}(x_1) \wedge -1\ lt^\sharp_{\mathbb{M}(l)}\ e \wedge e\ lt^\sharp_{\mathbb{M}(l)}\ len(x_1)\end{array}}{\Sigma \vdash^\dagger (x_1[e] := x_2)^l \Rightarrow \Sigma, \xi}$$

$$\frac{\begin{array}{c}\Sigma_1 \vdash^\ddagger e : int, \xi \\ \Sigma_1 \vdash^\dagger c_1 \Rightarrow \Sigma_2, \xi \quad \Sigma_1 \vdash^\dagger c_2 \Rightarrow \Sigma_2, \xi\end{array}}{\Sigma_1 \vdash^\dagger (if\ e\ c_1\ c_2)^l \Rightarrow \Sigma_2, \xi} \qquad \frac{\begin{array}{c}\Sigma_1 \vdash^{\mathbb{M}(l),nn^\sharp,lt^\sharp} e : int, \xi \\ \Sigma_1 \vdash^\dagger c \Rightarrow \Sigma_2, \xi \quad \Sigma_2 <: \Sigma_1\end{array}}{\Sigma_1 \vdash^\dagger (while\ e\ c)^l \Rightarrow \Sigma_1, \xi}$$

$$\frac{}{\Sigma \vdash^\dagger x := new(\tau[i]) \Rightarrow \Sigma[x \mapsto \tau[]], \top} \qquad \frac{}{\Sigma \vdash^\dagger skip \Rightarrow \Sigma, \top}$$

$$\frac{\Sigma_1 \vdash^\dagger c_1 \Rightarrow \Sigma_2, \xi \quad \Sigma_2 \vdash^\dagger c_2 \Rightarrow \Sigma_3, \xi}{\Sigma_1 \vdash^\dagger c_1; c_2 \Rightarrow \Sigma_3, \xi} \qquad \frac{\begin{array}{c}\Sigma_1 <: \Sigma_2 \quad \Sigma_2 \vdash^\dagger c \Rightarrow \Sigma_3, \xi_1 \\ \Sigma_3 <: \Sigma_4 \qquad \xi_1 <: \xi_2\end{array}}{\Sigma_1 \vdash^\dagger c \Rightarrow \Sigma_4, \xi_2}$$

Figure 6: Type Rules for Commands

Apart from the parts related to the parameterization, the expression and command type rules are entirely standard. With respect to the parameterization specifics, the type rules for array size, and array indexing make use of the parameterized information and occur in two forms: one that is able to exclude the possibility of exceptions, and one that is not.

For the array size operator it suffices to rule out that the variable x ever contains *nil* to rule out the possibility of exceptions, for array indexing (for both the expression and the command) we must demand that the index is greater or equal to zero, and that the index is smaller than the size of the array in addition to the demand that the variable is non-nil. For an example detailing the type derivation of a small program with different parameterized information see Appendix A.

$$\frac{}{\delta \vdash i : \mathit{int}} \qquad \frac{\delta(p) <: \tau}{\delta \vdash p : \tau} \qquad \frac{\delta \vdash v : \tau}{\delta \vdash v : \tau, \xi} \qquad \frac{}{\delta \vdash \bot : \tau, \bot_\Sigma}$$

$$\frac{\forall i \in \mathit{dom}(a) \;.\; \delta \vdash a[i] : \tau}{\delta \vdash a : \tau[]}$$

$$\frac{\forall p \in \mathit{dom}(\delta) \;.\; \delta \vdash h(p) : \delta(p)}{\delta \vdash h} \qquad \frac{\forall i \in \mathit{dom}(a) \;.\; \delta \vdash a[i] : \tau}{\delta \vdash a : \tau[]}$$

$$\frac{\forall x \in \mathit{dom}(\Sigma) \;.\; \delta \vdash s(x) : \Sigma(x)}{\delta \vdash s : \Sigma} \qquad \frac{\delta \vdash s : \Sigma \quad \delta \vdash h}{\delta \vdash (s, h) : \Sigma}$$

$$\frac{\delta \vdash E : \Sigma_2}{\delta \vdash^\dagger \bot_E : \Sigma_1, \bot_{\Sigma_2}} \qquad \frac{\delta \vdash E : \Sigma}{\delta \vdash^\dagger E : \Sigma, \xi} \qquad \frac{\Sigma_1 \vdash^\dagger c \Rightarrow \Sigma_2, \xi \quad \delta \vdash E : \Sigma_1}{\delta \vdash^\dagger \langle E, c \rangle : \Sigma_2, \xi}$$

Figure 7: Well-formedness

4.3 Correctness

With this we are ready to formulate correctness for the parameterized type system. As is standard we split the correctness argument into two theorems, *progress* — intuitively, that well-typed commands and expressions are able to execute in all environments that conform to the entry environment type of the command or expression — and *preservation* — intuitively, that the result of running the command or expression conforms to the exit type of the same. In contrast to the preservation proof, the progress proof is independent of the parameterized information. For space reasons we omit the progress proof.

Well-formedness The well-formedness relation in Figure 7 defines when values, environments and contexts are well-formed with respect to the corresponding types. The pointer typing δ is a map from pointers to record names, and makes the relation inductively definable in the presence of cyclic heaps. As above, $\vdash^\dagger$ is used as short notation for $\vdash^{\mathbb{M}, nn^\sharp, lt^\sharp}$. In short, a value is well-formed with respect to any exception type, whereas an error is only well-formed with respect to an exception type that indicates the possibility of the error, and similarly for well-formed environments, with the addition of the demand that the exception environment is well-formed in the exception environment type. A configuration is well-formed in the type Σ_2, ξ if there exists an environment type Σ_1 in which the environment E is well-formed such that the command is well-typed with Σ_1 as entry type and the Σ_2, ξ as exit type.

Preservation of Types of Expressions and Commands Preservation of types of expressions expresses that well-typed expressions preserve well-formedness under execution, i.e., for an expression e s.t. $\Sigma \vdash^{\mathbb{E}, nn^\sharp, lt^\sharp} e : \tau, \xi$ running e in Σ-well-formed environments that are modeled by the abstract environment $\mathbb{E}$ will result in τ, ξ-well-formed values.

Theorem 4.1 (Preservation of Types of Expressions)

$$\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e : \tau, \xi \Longrightarrow \forall E \in \gamma(\mathbb{E}) \,.\, \delta \vdash E : \Sigma \wedge \langle E, e\rangle \Downarrow v \Longrightarrow \delta \vdash v : \tau, \xi$$

Proof 4.1 *By induction on the derivation of* $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e : \tau, \xi$. *Intuitively, in each case, the proof proceeds by an inversion of* $\langle E, e\rangle \Downarrow v$, *which results in a number of sub-cases — one for each semantic rule for the expression, including the ones resulting in exceptions. However, in the cases where the type system can rule out exceptions it contains enough information about the execution from the use of the plugins on the abstract environment to prove the impossibility of an exception.*

We exemplify the difference between a standard proof and a parameterized proof by proving the correctness for the array indexing cases, corresponding to the two type rules for array indexing.

Assume (1) $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e : \tau, \xi$, (2) $E \in \gamma(\mathbb{E})$, (3) $\delta \vdash E : \Sigma$ *and* (4) $\langle E, e\rangle \Downarrow v$. *We must show* $\delta \vdash v : \tau, \xi$.

array indexing with exceptions *In this case the last applied type rule in the derivation is the rule that cannot rule out exceptions, which gives* (5) $\Sigma(x) = \tau[]$, $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e' : int, \xi'$, $\neg(nn^\sharp_{\mathbb{E}}(x) \wedge -1\ lt^\sharp_{\mathbb{E}}\ e' \wedge e'\ lt^\sharp_{\mathbb{E}}\ len(x))$, $\xi = \bot_\Sigma$ *and that* $e = x[e']$. *Inversion of* (4) *gives us the following four cases.*

1) **nil-pointer exception** *This case gives* $v = \bot$ *from which the result* $\delta \vdash \bot : \tau, \bot_\Sigma$ *is immediate.*

2) e **leads to an exception** *Same as the case above.*

3) **index out of bounds** *Same as the case above.*

4) **successful execution** *Let* $E = (s, h)$; *this case gives* (6) $s(x) = p$, $h(p) = (i_1, d)$, $\langle E, e'\rangle \Downarrow i_2$, (7) $i_2 \in [0..(i_1 - 1)]$ *and* $v = d(i_2)$. *From* $(3, 5, 6)$ *we get* $\delta \vdash p : \tau[]$, *which in turn gives* $\delta(p) <: \tau[]$, *which means* (8) $\delta(p) = \tau[]$, *since array subtyping is invariant. Further,* (3) *and* (8) *give* $\delta \vdash h(p) : \tau[]$, *which gives* $\forall i \in dom((i_1, d)) \,.\, \delta \vdash (i_1, d)(i) : \tau$. *Thus,* (7) *gives us that* $i_2 \in dom((i_1, d))$, *from which we get the result* $\delta \vdash d(i_2) : \tau$.

array indexing without exceptions *In this case the last applied type rule in the derivation is the rule that rules out exceptions, which gives* $\Sigma(x) = \tau[]$, $\Sigma \vdash^{\mathbb{E},nn^\sharp,lt^\sharp} e' : int, \xi$, (5) $nn^\sharp_{\mathbb{E}}(x)$, (6) $-1\ lt^\sharp_{\mathbb{E}}\ e'$, (7) $e'\ lt^\sharp_{\mathbb{E}}\ len(x)$, *and that* $e = x[e']$. *Again, inversion of* (4) *gives us the following four cases.*

1) **nil-pointer exception** *This case gives* (8) $s(x) = nil$. (1) *and* (5) *give* $\forall E \in \gamma(\mathbb{E})$. $x \in nn^\diamond_E$, *which together with* (2) *gives* (9) $\langle E, x\rangle \Downarrow nil \Longrightarrow nil \in nn$. (8) *gives* $\langle E, x\rangle \Downarrow nil$, *which together with* (9) *gives* $nil \in nn$ *which is a contradiction.*

2) e **leads to an exception** *This case gives* $\langle E, e'\rangle \Downarrow \bot$ *which together with the induction hypothesis gives* $\xi = \bot_\Sigma$ *from which the result is immediate.*

3) index out of bounds *This case gives* $s(x) = p$, $h(p) = (i_1, d)$, $\langle E, e'\rangle \Downarrow i_2$ *and* (8) $i_2 \notin [0..(i_1 - 1)]$. *In a way similar to 1) above, we use* (6) *to prove that it is impossible that* i_2 *is less than* 0 *and* (7) *to prove that it is impossible that* i_2 *is greater than or equal to* i_1. *Together this contradicts* (8) *and we have reached a contradiction.*

4) successful execution *This case is proven in the same way as case 4) in array indexing with exceptions.*

Thus, as the proof of preservation of types for array indexing shows, we achieve higher precision in the exception type by using the parameterized information to prove some cases impossible as described in Section 3.3. As discussed, the proof for the parameterized type system is essentially identical to the original proof where there is no parameterized information, with the difference that two cases are proved impossible.

Given the well-formedness formulation for configurations above, preservation of types of commands can be formulated in the same way as preservation of types of expressions.

Theorem 4.2 (Preservation of Types of Commands)

$$\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c \Rightarrow \Sigma_2, \xi \wedge \mathit{entrysol}_c^{\mathcal{C}}(\mathbb{M}) \Longrightarrow$$
$$\forall E \in \mathcal{C} \,.\, \delta_1 \vdash E : \Sigma_1 \wedge \langle E, c\rangle \rightarrow C \Longrightarrow \exists \delta_2 \,.\, \delta_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} C : \Sigma_2, \xi$$

Proof 4.2 *By induction on the derivation of* $\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c \Rightarrow \Sigma_2, \xi$. *Intuitively, such an induction proceeds by a case analysis of the type rules; in all cases* $\mathit{entrysol}_c^{\mathcal{C}}(\mathbb{M})$ *guarantees that* $\mathbb{M}$ *is a sound representation of the possible executions of* c *for (at least) all environments in* $\mathcal{C}$. *As above, inversion of* $\langle E, c\rangle \rightarrow C$ *gives rise to cases of abnormal execution that can be proved impossible in the cases where the type system rules out exception by using fact that* $nn^\sharp$ *and* $lt^\sharp$ *are sound approximations for the non-nil and less-than plugin properties respectively.*

We consider a representative case: the proof for sequence. The proof for array update follows the structure of the proof of array indexing above, with the difference that the proof of update relies on a standard proof for array updates in the cases of successful execution; the remaining cases are either standard or subject to similar changes.

Assume $\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c \Rightarrow \Sigma_2, \xi$, (1) $\mathit{entrysol}_c^{\mathcal{C}}(\mathbb{M})$, $E \in \mathcal{C}$, (3) $\delta_1 \vdash E : \Sigma_1$ *and* (4) $\langle E, c\rangle \rightarrow C$. *We must show* $\exists \delta_2 \,.\, \delta_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} C : \Sigma_2, \xi$.

sequence *In this case the last applied rule in the derivation is the rule for sequence, which gives* (5) $\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c_1 \Rightarrow \Sigma, \xi_1$, (6) $\Sigma \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c_2 \Rightarrow \Sigma_2, \xi_2$, $c = c_1; c_2$ *and* $\xi = \xi_1 \sqcup \xi_2$ *for some intermediate* Σ, ξ_1 *and* ξ_2. *Inversion of* (4) *gives the following three cases.*

skip *This case gives* $c_1 = skip$ *and* $C = \langle E, c_2\rangle$. *From* (5) *we get* $\Sigma = \Sigma_1$ *and* $\xi_1 = \top$. *The definition of the operator* $\sqcup$ *gives* $\xi = \xi_2$. *This, together with* (3) *and* (6) *gives the result.*

context reduction *This case gives* $c = R[c_1']$ *and* $\langle E, c_1'\rangle \rightarrow \langle E', c_2'\rangle$. *The case where* $c_1' = c$ *is identical to the case skip above. We, therefore, only consider the case where* $c_1' \neq c$. *This gives* $R = R'[c_1']; c_2$ *and* $C = \langle E', R'[c_2']; c_2\rangle$. *Since the execution of* c_1 *is a prefix of the execution of* $c_1; c_2$ *we get* $entrysol^{\mathcal{C}}_{c_1}(\mathbb{M})$ *from* (1). *By applying the induction hypothesis we get* (7) $\Sigma' \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} R'[c_2'] \Rightarrow \Sigma, \xi_1$ *and* (8) $\delta_2 \vdash E' : \Sigma'$ *for some* Σ', *and some* δ_2. *The typing rule for sequences applied to* (6) *and* (7) *gives* $\Sigma' \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} R'[c_2']; c_2 \Rightarrow \Sigma_2, \xi_1 \sqcup \xi_2$ *which together with* (8) *gives the result.*

exception *This case gives* $c = R[c_1']$ *and* $\langle E, c_1'\rangle \rightarrow \langle E', c_2'\rangle$. *Once again we only consider the case where* $c_1' \neq c$, *which gives* $R = R'[c_1']; c_2$ *and* $C = \bot_{E'}$. *From here on the result is immediate from the induction hypothesis, which can be applied using the same reasoning as in the previous case.*

Top-level Correctness of Commands The proof of top-level correctness of commands makes use of progress of commands, i.e., that well-type programs are able to take one step of execution in correspondingly well-formed environments.

Theorem 4.3 (Progress of Commands)

$$\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c \Rightarrow \Sigma_2 \Longrightarrow \forall E\ .\ \delta \vdash E : \Sigma_1 \Longrightarrow \exists C\ .\ \langle E, c\rangle \rightarrow C$$

Proof 4.3 *By induction on* $\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c \Rightarrow \Sigma_2$.

Let $\langle E, c\rangle \rightarrow^n C$ be the obvious lifting of the small step evaluation to evaluation of n consecutive steps. With this we are ready to formulate the top-level correctness of commands, that well-typed commands terminate in a well-formed environment or result in well-formed configurations regardless of the number of execution steps. For convenience we let T range over terminal configurations.

Theorem 4.4 (Top-level Correctness of Commands)

$$\begin{aligned}\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c_1 \Rightarrow \Sigma_2, \xi \wedge entrysol^{\mathcal{C}}_{c_1}(\mathbb{M}) \Longrightarrow \forall E_1 \in \mathcal{C}\ .\ \delta_1 \vdash E_1 : \Sigma_1 \Longrightarrow \\ \forall n.\ (\exists n' \leq n, T, \delta_2.\ \langle E_1, c_1\rangle \rightarrow^{n'} T \wedge \delta_2 \vdash T : \Sigma_2, \xi) \vee \\ (\exists E_2, c_2, \delta_2.\ \langle E_1, c_1\rangle \rightarrow^n \langle E_2, c_2\rangle \wedge \delta_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} \langle E_2, c_2\rangle : \Sigma_2, \xi)\end{aligned}$$

Proof 4.4 *Assume* (1) $\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c_1 \Rightarrow \Sigma_2, \xi$, (2) $entrysol^{\mathcal{C}}_{c_1}(\mathbb{M})$, (5) $\delta_1 \vdash E_1 : \Sigma_1$ *for some* $E_1 \in \mathcal{C}$ *and* δ_1. *We proceed by induction on* n.

base case *This case gives* $n = 0$. (1) *and* (5) *give* $\delta_1 \vdash \langle E_1, c_1\rangle : \Sigma_2, \xi$ *which together with* $\langle E_1, c_1\rangle \rightarrow^0 \langle E_1, c_1\rangle$ *gives the result.*

induction step *In this case we show that the property holds for* $n+1$ *assuming that it holds for* n*. Case analysis over the applied induction hypothesis gives the following two cases.*

terminated reduction *This case gives* $\langle E_1, c_1\rangle \rightarrow^{n'} T$ *and* $\delta_2' \vdash T : \Sigma_2, \xi$ *for some* $n' \leq n$*,* T *and* δ_2'*. The result follows immediately from* $n' \leq n+1$*.*

non-terminated reduction *This case gives* $\langle E_1, c_1\rangle \rightarrow^{n} \langle E_2', c_2'\rangle$ *and* (6) $\delta_2' \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} \langle E_2', c_2'\rangle : \Sigma_2, \xi$ *for some* E_2'*,* c_2' *and* δ_2'*. From* (6) *we get* (7) $\Sigma_1' \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c_2' \Rightarrow \Sigma_2, \xi$ *and* (8) $\delta_2' \vdash E_2' : \Sigma_1'$ *for some* Σ_1'*. From Theorem 4.3, together with* (7) *and* (8) *we get* (9) $\langle E_2', c_2'\rangle \rightarrow C$ *for some* C*, which gives* (10) $\langle E_1, c_1\rangle \rightarrow^{n+1} C$*. Lemma 3.1, Preservation of Entry Solutions under Execution, and* (2) *gives* (10) $entrysol_{c_2'}^{\mathcal{C}'}(\mathbb{M})$ *for a* $\mathcal{C}'$ *such that* $E_2' \in \mathcal{C}'$*. Theorem 4.2, Preservation of Types of Commands, applied to* (7)*,* (10)*,* (3)*,* (4)*,* (8) *and* (9) *gives* $\delta_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} C : \Sigma_2, \xi$ *for some* δ_2*. The result follows immediately from this and* (10)*.*

5 Related Work

The method presented in this paper combines an analysis, formulated as a type system, with a number of external analyses, computing information useful to the type system, by parameterizing the type system over the computed information.

Similar in spirit is the work by Foster, Fähndrich and Aiken [FFA99] in which they present a framework for augmenting existing type systems with type qualifiers, e.g. `const` and `nonnil`. Our work differs from theirs in that they provide a framework to compute the qualifiers, rather than making use of them.

In [CMM05] Chin, Markstrum and Millstein investigate a method for supporting user-defined semantic type qualifiers that are closely related to unary plugins. As above, their work is aimed at computing an analysis result, rather than modularly making use of it. In addition to reason about soundness they propose a method to automatically verify the soundness of the extension using an automatic theorem prover.

Among the type systems making use of additional information are type systems that eliminate array bound checks, e.g., [XP98], using a decidable formulation of dependent types. It should be pointed out that even though the type checking is decidable the inference is not; nothing in our approach rules out inference. In [HS06] Hedin and Sands use a simplistic type based inference of nil-pointers needed to allow the use of non-secret fields in objects pointed to by pointers with secret pointer values. We believe that the clarity, correctness proof and power of their system could benefit greatly by being reformulated in our framework.

In [CW00] Crary and Weirich present a type system for resource bound verification, e.g., memory usage and execution time. Their type system goes

beyond the capacity of the plugins framework — time and memory usage are not values in a standard semantics. It could potentially be interesting to see to what extent the plugins model can be modified to encompass such extensions.

While this work suggests resolving type errors by using more and more elaborate parameterized analyses, Flanagan [Fla06] suggests pushing checks that cannot be statically resolved to runtime checks, cf. type cast checks in Java. For many uses of the plugins framework, uniting the two approaches could prove beneficial — if the program cannot be statically proven correct using a different external analysis, Flanagan's method could be applied to insert a dynamic check.

With respect to other work on combining static analyses, if the analyses we want to combine are formulated as abstract interpretations, a number of techniques from the large body of work on abstract interpretation [CC77, CC79, GT06] becomes applicable. An example of such a combination is the reduced product method. Similar to our method, the combination can be done in a systematic way and correctness of the resulting analysis follows from correctness of the combined analyses.

An advantage of the abstract interpretation framework is that for partially overlapping analyses and a combination like the reduced product, the analyses will benefit from each other. Each analysis can make use of the information computed by the other analyses, which stands in contrast to our method where the external analyses cannot make direct use of the derivation of the parameterized type system.

However, an obvious restriction of the abstract interpretation framework is that all analyses must be formulated as abstract interpretations, which is not always the case. Reformulating, for example, a type based analysis into an abstract interpretation is not always easily done nor desirable, as for example indicated by the field of security where the analyses tend to be type based [SM03]. Our approach does not have that restriction. A type system can be combined with any external analyses that compute valid solutions. If the external analyses are formulated as abstract interpretations our method can be combined with the abstract interpretation framework to make use of, for example, reduced products.

6 Conclusions and Future Work

We have presented a method for parameterizing program analyses for imperative small step semantics with information about the programs' execution. The appeal of the method compared to approaches where additional information about the programs' execution is provided by extending the type system with capabilities of computing the additional information, i.e., fusing the type system with another analysis, lies in that:

- The parameterization does not impose heavy changes to the type system. The rules remain relatively close to the original rules; only the use of the additional information is added to the rules where the information is used

— other rules remain essentially unaffected. Comparatively, fusing an analysis modifies all rules to compute the information, in addition to the uses of the information in certain rules.

- The parameterization gives the possibility of changing the parameterized analysis with relative ease — proofs for the family of analyses[1] and decision procedures with corresponding soundness proofs have to be done. Comparatively, changing the analysis for a fused type system means creating a new fused type system and correctness proof from scratch.

The method is based on the identification of a generic format for information exchange between the program analysis and the parameterized results, together with methods — the plugins — for asking specific questions about the each program parts execution environment.

To exemplify the method we have given an overview of the steps involved in parameterizing an existing type system, including the changes to the type system itself, but also the changes to the correctness proof of the type system. A corner stone in this work is the attempt to make the correctness proof a natural part of the parameterization process so that the proof burden for each parameterization is relatively low.

A drawback is that the resulting system may no longer be compositional; e.g. a compositional type system becomes non-compositional if the parameterized information is not compositional. Another restriction is that the parameterization is one-way only; there is no back propagation of type information that could have been used by the parameterized analysis.

Future Work The motivation for this work grew out of a perceived need to increase the precision of type based analyses of secure information flow. For this reason a natural continuation of this work is to apply the method to an information flow type system.

In addition to this, an implementation of the parameterized type system of this paper would be valuable to asses the practicality of the approach. Of particular interest would be to implement a staged type system, where the reason for a type failure is analyzed and given as feedback to the next stage. The benefit of doing this is apparent in cases where the abstract environment map is a combination of the result of a number of external analyses. One way to view a set of increasingly precise external analyses is as a matrix with one dimension for each type of analysis and plugin property. In the general setting where a parameterized type system uses multiple external analyses the external analyses build up a multi-dimensional matrix where each point corresponds to a particular instantiation of the type system.

Acknowledgements This work was partly supported by the Swedish research agencies SSF, VR and Vinnova, and by the Information Society Technologies

[1] The proof only has to be done once for each family, and typically includes a way of converting the analysis information provided by the family to the format of the parameterization.

programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

Bibliography

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 91–102, New York, NY, USA, 2006. ACM Press.

[BPR07] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In R. De Niccola, editor, *European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CMM05] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 40(6):85–95, 2005.

[CW00] Karl Crary and Stephnie Weirich. Resource bound certification. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–198, New York, NY, USA, 2000. ACM.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *Programming Language Design and Implementation (PLDI)*, 34(5):192–203, 1999.

[Fla06] Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, New York, NY, USA, 2006. ACM.

[GH08] Tobias Gedell and Daniel Hedin. Plugins for structural weakening and strong updates. Technical Report 2008:13, Computing Science Department, Chalmers, 2008.

[GT06] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 376–386, New York, NY, USA, 2006. ACM Press.

[HS06] David Hedin and David Sands. Noninterference in the presence of non-opaque pointers. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.

[HS08] Sebastian Hunt and David Sands. Just forget it – the semantics and enforcement of information erasure. In *Programming Languages and Systems. 17th European Symposium on Programming, ESOP 2008*, number 4960 in LNCS, pages 239–253. Springer Verlag, 2008.

[Pie02] Benjamin C. Pierce, editor. *Types and Programming Languages*. MIT Press, 2002.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, 33(5):249–257, 1998.

A Examples

This section considers a small example, that highlights some of the properties of the above type system. In particular, we show the staged nature of the type system and how it reverts to a standard type system when instantiated with a void analysis — an analysis where the decision procedures are constant false functions. First, consider the following simple program.

```
a := new(int[10]);
i = 0;
while (i < size(a)) {
    a[i] := 0;
    i = i + 1;
}
```

The program allocates a new array of integers and initializes it to all zeroes. There are two sources of exceptions in this program: 1) the size operation throws an exception if a is nil, and 2) the assignment to the array throws an exception if a is the nil-pointer or if i is out of bounds. A typical standard type system would not be able to exclude the possibilities of exceptions in this program, even though it is relatively easy to extend a standard type system to exclude exceptions in this particular case. The main point of the following is not to form an analysis that is able to deal with the above example, but rather to exemplify the operation of the type system.

Below, we consider type derivations of the above program using three different external analyses: a void analysis, a simplistic non-nil analysis, and an extension to the non-nil analysis with an array index analysis — simply pairing the results of the non-nil and an array index analysis is enough. We will not present any details of the analyses, but rather just appeal to the reader's intuition that analyses that can handle the above example are relatively simple to form.

Void Analysis A void analysis is an analysis where all decision procedures are constant false. Clearly, when instantiated with a void analysis, the type system reverts to a standard type system. Let c refer to a labeled version of the above program.

$$c \equiv a := new(int[10]); (i := 0)^{l_1}; (while\ i < len(a)\ \{(a[i] := 0)^{l_3}; (i := i + 1)^{l_4}\})^{l_2}$$

Second, to fit the derivation onto one page we analyze and number the type derivations for the individual statements of the program in Figure 8 below. For brevity the derivation of $i + 1$ in derivation (6) is done in one step. We let $\Sigma_1 = [a \mapsto int[]]$, $\Sigma_2 = [a \mapsto int[], i \mapsto int]$. The grayed premises in derivation (3), and (5), mark uses of the plugins procedures; clearly, a sound void analysis cannot rule our any environments making the following derivation the most

1
$$\frac{}{\emptyset \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} a := new(int[10]) \Rightarrow \Sigma_1, \top}$$

2
$$\frac{\Sigma_1 \vdash^{\mathbb{M}(l_1),nn^\sharp,lt^\sharp} 0 : int, \top}{\Sigma_1 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} (i := 0)^{l_1} \Rightarrow \Sigma_2, \top}$$

3
$$\frac{\Sigma_2(a) = int[] \qquad \neg nn^\sharp_{\mathbb{M}(l_2)}(a)}{\Sigma_2 \vdash^{\mathbb{M}(l_2)} len(a) : int, \bot_{\Sigma_2}}$$

4
$$\frac{\dfrac{\Sigma_2(i) = int}{\Sigma_2 \vdash^{\mathbb{M}(l_2),nn^\sharp,lt^\sharp} i : int, \top} \qquad [3]}{\Sigma_2 \vdash^{\mathbb{M}(l_2),nn^\sharp,lt^\sharp} i < len(a) : int, \bot_{\Sigma_2}}$$

5
$$\frac{\dfrac{\Sigma_2(i) = int}{\Sigma_2 \vdash^{\mathbb{M}(l_3),nn^\sharp,lt^\sharp} i : int, \top} \qquad \Sigma_2(a) = int[] \qquad \Sigma_2(i) = int \qquad int <: int \qquad \neg(nn^\sharp_{\mathbb{M}(l_3)}(a) \wedge -1 \ lt^\sharp_{\mathbb{M}(l_3)} \ i \wedge i \ lt^\sharp_{\mathbb{M}(l_3)} \ len(a))}{\Sigma_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} (a[i] := 0)^{l_3} \Rightarrow \Sigma_2, \bot_{\Sigma_2}}$$

6
$$\frac{\Sigma_2 \vdash^{\mathbb{M}(l_4),nn^\sharp,lt^\sharp} i + 1 : int, \top}{\Sigma_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} (i := i + 1)^{l_4} \Rightarrow \Sigma_2, \top}$$

7
$$\frac{[5] \qquad [6]}{\Sigma_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} (a[i] := 0)^{l_3}; (i := i + 1)^{l_4} \Rightarrow \Sigma_2, \bot_{\Sigma_2}}$$

8
$$\frac{[4] \qquad [7] \qquad \Sigma_2 <: \Sigma_2}{\Sigma_2 \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} (while \ i < len(a) \ \{(a[i] := 0)^{l_3}; (i := i + 1)^{l_4}\})^{l_2} \Rightarrow \Sigma_2, \bot_{\Sigma_2}}$$

Figure 8: Subderivations

precise possible.

$$\frac{[1] \qquad \dfrac{[2] \qquad [8]}{\ldots}}{\emptyset \vdash^{\mathbb{M},nn^\sharp,lt^\sharp} c \Longrightarrow \Sigma_2, \bot_{\Sigma_2}}$$

Simple Non-nil Analysis It is easy to envision an analysis that is able to detect that $a \neq nil$ after the first line; given such an analysis the type system would be able to rule out exceptions in the controlling expression of the while $i < len(a)$, but not in the array assignment $(a[i] := 0)^{l_3}$, since the analysis cannot see that the index is within bounds. Given the result $\mathbb{M}$ of such an analysis and a sound non-nil decision procedure $nn^\sharp$ it is clear that $nn^\sharp_{\mathbb{M}(l_3)}(x)$

and $nn^\sharp_{\mathbb{M}(l_4)}(x)$ hold. Thus, derivation (3) above can be replaced by

$$\frac{\Sigma_2(a) = int[] \qquad nn^\sharp_{\mathbb{M}(l_2)(a)}}{\Sigma_2 \vdash^{\mathbb{M}(l_2)} len(a) : int, \top}$$

This, however, does not affect the resulting exception type of c, since we do not differentiate between different sources of exceptions; if we did, we would — with some minor changes to the type rule for array assignment — be able to rule out the possibility on nil-pointer exceptions in c.

Simple Array Index Analysis To rule out the presence of exceptions in c entirely some kind of array bounds analysis is needed in addition to the nil-pointer analysis; for c even rather simplistic analyses are able to see that i is always within the bounds of the array pointed to by a. We form a new analysis by pairing the nil-pointer analysis with the array bound analysis; the result of the new analysis $\mathbb{M}$ is a map from labels to pairs of abstract environments—one from each analysis, $nn^\sharp$ is left projection composed with the decision procedure for the non-nil analysis, and similarly $lt^\sharp$ is right projection composed with the decision procedure for the array bounds analysis. Given an environment map $\mathbb{M}$ of the combined analysis, a sound non-nil decision procedure $nn^\sharp$ and a sound array bound decision procedure $lt^\sharp$, it is clear that $nn^\sharp_{\mathbb{M}(l_3)}(x)$, and $nn^\sharp_{\mathbb{M}(l_4)}(x) \wedge -1\ lt^\sharp_{\mathbb{M}(l_4)}\ 0 \wedge 0\ lt^\sharp_{\mathbb{M}(l_4)}\ len(a)$ hold. Thus, as above, derivation (3) above can be replaced by

$$\frac{\Sigma_2(a) = int[] \qquad nn^\sharp_{\mathbb{M}(l_2)(a)}}{\Sigma_2 \vdash^{\mathbb{M}(l_2)} len(a) : int, \top}$$

and derivation (5) can be replaced with

$$\frac{\dfrac{\Sigma_2(i) = int}{\Sigma_2 \vdash^{\mathbb{M}(l_3), nn^\sharp, lt^\sharp} i : int, \top} \qquad \Sigma_2(a) = int[] \qquad \begin{array}{c} \Sigma_2(i) = int \\ nn^\sharp_{\mathbb{M}(l_3)}(a) \end{array} \qquad \begin{array}{c} int <: int \\ -1\ lt^\sharp_{\mathbb{M}(l_3)}\ i \end{array} \qquad \begin{array}{c} \\ i\ lt^\sharp_{\mathbb{M}(l_3)}\ len(a)) \end{array}}{\Sigma_2 \vdash^{\mathbb{M}, nn^\sharp, lt^\sharp} (a[i] := 0)^{l_3} \Rightarrow \Sigma_2, \bot_{\Sigma_2}}$$

Now, since the the array size expression $len(a)$ in derivation (3), and the array assignment $(a[i] := 0)^{l_3}$ in derivation (5) were the only sources of exceptions the following derivation is valid.

$$\frac{[1] \qquad \dfrac{[2] \qquad [8]}{\ldots}}{\emptyset \vdash^{\mathbb{M}, nn^\sharp, lt^\sharp} c \Longrightarrow \Sigma_2, \top}$$

where the exception types of derivation (4), (7) and (8) are changed accordingly. Thus, we see how the additional information about the program's execution — nil-pointer information, and array bound information — is used in the derivation of an exception free type of c.

Plugins for Structural Weakening and Strong Updates

Tobias Gedell Daniel Hedin

Abstract

We present a general way of making use of may- and must-alias information to achieve flow-sensitive type systems that allow for flow-sensitivity on the heap. In particular, we show how may-alias information can be used for a limited form of flow-sensitivity — *structural weakening* — that allows type changes on the heap that are compatible with the subtype hierarchy. Further, we show how the combination of may- and must-alias information can be used to achieve *strong updates*, i.e., type changes on the heap that are not compatible with the subtype hierarchy, resembling the typical type rule for updates of variables in flow sensitive type systems. This work has been enabled by the use of our plugin framework — a framework for parameterizing type systems over the results of abstract interpretations — that allows us to abstract away from the computation of the alias information. In addition, our successful use of the plugin mechanism to extract both may- and must-alias information shows its strength.

1 Introduction

One dimension of program analyses is flow-sensitivity; the result of a flow-sensitive analysis depends on the order of the instructions of a program, whereas the result of a flow-insensitive analysis does not. Frequently, flow-sensitive analyses achieve higher precision than flow-insensitive.

Even though type systems are predominantly flow-insensitive, flow-sensitive type systems arise naturally, as shown by, for example, linear and affine type systems [Pie05]. A more well-known example of a flow-sensitive type system is the type system of Java bytecode where the limited number or registers forces (from a practical standpoint) the type system to be able to change the types of registers; each instruction is typed in a pre- and a post-type, and the assignment instruction changes the type of the target register in the post type to the type of the source value.

The static flow-sensitivity of Java bytecode is limited to registers for a reason; aliasing prohibits the flow-sensitivity to easily extend to the heap. To be able to change types on the heap, e.g., the type of a field of a certain object, it must be made sure that the types of all aliased locations are changed, otherwise a way of

freely casting between the types by writing into one location and reading from another is introduced. Thus, for instance, it is not safe to extend the subtyping relation to arrays, i.e., to say that `String[]` is a subtype of `Object[]` because `String` is a subtype of `Object`, since that would allow casting from the type `Object` to the type `String`.[1]

The standard solution for this is to enforce *invariant typing* for all heap locations, involving both prohibiting type changes based on updates, and restraining the subtype relation to *invariant subtyping* for arrays, and *width subtyping* for objects [Pie02].

From a practicality standpoint, width subtyping is good enough for "standard" types. For information flow security it is not necessarily the case. Whereas the need to freely change the types of parts of the heap may seem far fetched, i.e., from a boolean to an integer, the need to change a location from holding public integers to holding secret integers is not as unreasonable, especially considering that, unlike standard types, information flow types are intrinsically flow sensitive. In addition to this, there are important extensions to basic information-flow security that rely on flow-sensitivity [HS08].

Contribution We present a general method for making use of may- and must-alias information to allow for flow-sensitive types on the heap. Whereas previous work has tried to combine the computation of the alias information and its usage, we use our recently proposed abstract-interpretation plugin framework [GH08] to decouple the computation and the usage of the alias information. This allows us to use the alias information in a general and clear way, while at the same time allowing us to instantiate the resulting type system with different alias analyses — from the most basic ones to, e.g., the most elaborate shape analyses.

The main contributions of this paper are that we 1) show how the plugin framework can be used to carry over structural information about the heap, 2) show how may-alias information can be used to formulate a structural subtyping rule, and 3) show how may- and must-alias information can be used in combination to allow for strong updates on the heap, i.e., updates that do not follow the subtype hierarchy.

Outline The paper is laid out as follows. Section 2 introduces the language, Section 3 recapitulates the needed parts of the method of parameterization, and Section 4 introduces the basic type system and the correctness statements — in particular *preservation of types*. Section 5 discusses the basic idea of flow-sensitive heap types, defines the used representation of may- and must-alias information — *structural environments* — and their semantics, and shows how structural heaps can be extracted from the parameterized alias information. Section 6 contains the first use of may-alias information to achieve a limited form of flow-sensitive heap types. The basic idea is to use the may-aliases to form a structural subtyping rule, where all symbolic locations that may be aliased are guaranteed to have the same type view, which guarantees that all

[1] Java *does* allow this, which forces a runtime check when storing objects into arrays.

concrete environments will be width well-formed. Section 7 shows how may- and must-alias information can be combined to support strong updates if the updated location is in an isolated cluster of must-pointers. Finally, Section 8 discusses related work, and Section 9 concludes.

2 Language

The language used to illustrate our method is a small imperative language with records.

Syntax Let f range over field names, b range over booleans, i range over integers, and x range over variable names. The syntax of the language is defined as follows, where A ranges over record type names.

$$\begin{array}{lrll} \text{Expressions} & e & ::= & \mathit{nil} \mid b \mid i \mid x \mid e_1 \star e_2 \mid x.f \\ \text{Commands} & c & ::= & x := e \mid x_1.f := x_2 \mid \mathit{if}\ e\ c_1\ c_2 \mid c_1; c_2 \mid \\ & & & \mathit{while}\ e\ c \mid x := \mathit{new}(A) \mid \mathit{skip} \end{array}$$

Values The environments are pairs of stores s and heaps h. The stores are maps from variables x to values v, and the heaps are maps from pointers p to records r. The records are maps from field names to values. Finally, the values are made up by booleans, integers and pointers; the *error lifted values*, ranged over by $v_\perp$, are either values or $\perp$ indicating an error. We impose the restriction that heaps may not associate the null-pointer to anything, i.e., the null-pointer must not be in the domain of any heap.

$$\begin{array}{lll} v & ::= & b \mid i \mid p \\ E & ::= & (s, h) \end{array} \qquad \begin{array}{lll} r & ::= & \{f_1 \mapsto v_1, \ldots, f_n \mapsto v_n\} \\ s & ::= & \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \\ h & ::= & \{p_1 \mapsto r_1, \ldots, p_n \mapsto r_n\} \end{array}$$

Let $r.f$ denote $r(f)$, and for $E = (s, h)$, let $E(x)$ denote $s(x)$, $E[x \mapsto v]$ denote $(s[x \mapsto v], h)$, $E(p)$ denote $h(p)$, and similarly for other operations on environments including variables or pointers.

Semantics We assume a simple reduction semantics for expressions of the form $\langle E, e\rangle \Downarrow v_\perp$.

The semantics of commands is given in terms of a small step semantics between configurations with transitions of the form $\langle E, c\rangle \to C$, where C is either one of the terminal configurations $\perp_E$ and E indicating abnormal and normal termination in the environment E, respectively, or a non-terminal configuration $\langle E, c\rangle$. Figure 1 contains the semantic rules for commands, where $\mathit{rec}(A)$ creates a fresh record of type A with all fields set to 0 or nil, depending on their type — we assume the existence of a map Δ from record type names to structural record types (defined in Section 4 below). The origin of this map is not significant for this work, and, thus, left unspecified, but is typically created from the program

$$\frac{}{\langle E, x := v\rangle \to \langle E[x \mapsto v], \mathit{skip}\rangle} \qquad \frac{s(x_1) = \mathit{nil}}{\langle (s,h), x_1.f := x_2\rangle \to \bot_{(s,h)}}$$

$$\frac{s(x_1) = p \qquad h(p) = r \qquad s(x_2) = v}{\langle (s,h), x_1.f := x_2\rangle \to \langle (s, h[p \mapsto r[f \mapsto v]]), \mathit{skip}\rangle}$$

$$\frac{}{\langle E, \mathit{if}\ \mathit{true}\ c_1\ c_2\rangle \to \langle E, c_1\rangle} \qquad \frac{}{\langle E, \mathit{if}\ \mathit{false}\ c_1\ c_2\rangle \to \langle E, c_2\rangle}$$

$$\frac{}{\langle E, \mathit{while}\ e\ c\rangle \to \langle E, \mathit{if}\ e\ (c; \mathit{while}\ e\ c)\ \mathit{skip}\rangle}$$

$$\frac{r = \mathit{rec}(A) \qquad p \notin \mathit{dom}(h)}{\langle (s,h), x := \mathit{new}(A)\rangle \to \langle (s[x \mapsto p], h[p \mapsto r]), \mathit{skip}\rangle}$$

$$\frac{}{\langle E, \mathit{skip}; c\rangle \to \langle E, c\rangle}$$

Figure 1: Semantic rules for commands

source, possibly in combination with a system specific map. The record name map is invariant under the execution of the program, and is left implicit in the rest of this paper.

Following [GH08] we extend the command language with label annotations to track how environments flow in and out of commands during execution and add the following rules for reduction of labeled commands. Let l range over labels drawn from the set of labels $\mathcal{L}$. A command c can be annotated with an entry label $(c)^l$, an exit label $(c)_l$, or both $(c)^{l_1}_{l_2}$. In the following, c ranges over possibly annotated commands, i.e., $(c)^l$ denotes a command with at least an entry label l, and similarly for exit labels. For while-loops decorated with an entry label we add the following reduction rule.

$$\frac{}{\langle E, (\mathit{while}\ e\ c)^l\rangle \to \langle E, (\mathit{if}\ e\ (c; (\mathit{while}\ e\ c)^l)\ \mathit{skip})^l\rangle}$$

For commands decorated with entry labels that are not while-loops, and for *skip* decorated with an exit label we add the following reduction rules.

$$\frac{\langle E_1, c_1\rangle \to \langle E_2, c_2\rangle}{\langle E_1, (c_1)^l\rangle \to \langle E_2, c_2\rangle} \qquad \frac{}{\langle E, (\mathit{skip})_l\rangle \to \langle E, \mathit{skip}\rangle}$$

As is common for small step semantics we use evaluation contexts R to determine the position of the next computation step.

$$R \quad ::= \quad \cdot \mid x := R \mid \mathit{if}\ R\ c\ c \mid R; c \mid (R)_l$$

The accompanying standard reduction rules, found in Figure 2, allow for left-most reduction of sequences, error propagation, reduction of expressions inside commands and reduction of commands under exit labels.

$$\frac{\langle E, e\rangle \Downarrow v}{\langle E, R[e]\rangle \to \langle E, R[v]\rangle} \qquad \frac{\langle E, e\rangle \Downarrow \bot}{\langle E, R[e]\rangle \to \bot_E}$$

$$\frac{\langle E_1, c_1\rangle \to \langle E_2, c_2\rangle}{\langle E_1, R[c_1]\rangle \to \langle E_2, R[c_2]\rangle} \qquad \frac{\langle E, c\rangle \to \bot_E}{\langle E, R[c]\rangle \to \bot_E}$$

Figure 2: Semantic Rules for Contexts

3 Parameterization

Before presenting the type system we recapture the fundamental parts of the method of parameterization. For a more thorough exposition see [GH08].

Abstract Environment Maps Let $\mathbb{E}$ range over some form of abstract environments with an associated concretization function γ, mapping abstract environments to sets of concrete environments. An abstract environment map is a map from labels to abstract environments. We say that an abstract environment map $\mathbb{M}$ is an entry/exit solution with respect to a command c_1 and a concrete environment E_1 if it represents all environments flowing into/out of each command as follows where *is* is the predicate for entry solutions and *os* the predicate for exit solutions.

$$is_{c_1}^{E_1}(\mathbb{M}) \equiv \forall E_2, c_2.\ \langle E_1, c_1\rangle \to^* \langle E_2, R[(c_2)^l]\rangle \Longrightarrow E_2 \in \gamma(\mathbb{M}(l))$$

$$os_{c_1}^{E_1}(\mathbb{M}) \equiv \forall E_2.\ \langle E_1, c_1\rangle \to^* \langle E_2, R[(skip)_l]\rangle \Longrightarrow E_2 \in \gamma(\mathbb{M}(l))$$

The definition is lifted to sets of initial environments $\mathcal{C}$ in the obvious way.

Plugins Properties and Plugins A *plugin property* $R^\diamond$ is a family of expression liftings of a relation R on values, defined in the following way.

$$(e_1, \ldots, e_n) \in R_E^\diamond \equiv \langle E, e_1\rangle \Downarrow v_1 \wedge \cdots \wedge \langle E, e_n\rangle \Downarrow v_n \Longrightarrow (v_1, \ldots, v_n) \in R$$

Plugin properties define the meaning of the *plugins*, in the sense that the plugins are approximations of the plugin properties.

A plugin is a family of relations on expressions indexed by abstract environments. $R^\sharp$ is said to be a plugin for $R^\diamond$ given that the following property holds.

$$(e_1, \ldots, e_n) \in R_{\mathbb{E}}^\sharp \Longrightarrow \forall E \in \gamma(\mathbb{E}).\ (e_1, \ldots, e_n) \in R_E^\diamond$$

The plugin framework cannot be used to directly transfer alias information. In Section 5 we show algorithmically how equality and inequality information about pointers can be used to build may- and must-alias views of the heap, respectively. Thus, in this paper we will be using two plugins: one for may-alias extraction corresponding to lifted pointer inequality, and one for must-alias

extraction corresponding to lifted pointer equality (excluding the null-pointer). Let $\mathcal{R}^{\neq}$ denote plugins for pointer inequality, and let $\mathcal{R}^{-}$ denote plugins for pointer equality. For convenience, let $\mathcal{R}^{+}$ denote the negation of the pointer inequality plugin — two pointers that cannot be shown to be unequal must be assumed to be aliased.

4 Type System

The type system introduced in this section is used as the base for two different extensions: structural weakening and strong updates which we investigate separately in Section 6 and Section 7 respectively.

Type Language The primitive types, ranged over by τ, are the type of booleans *bool*, the type of natural numbers *nat*, the type of integers *int*, and the pointer types, represented by record type names A. The record types ω are maps from fields to primitive types. As mentioned above, Δ is a map from record type names to record types. The store types, ranged over by Σ, are maps from variables to primitive types. The exception types, ranged over by ξ, are $\bot_\Sigma$, indicating the possibility that an exception is thrown in the environment type Σ, and $\top$ indicating that no exception is thrown. This is a simplification from typical models of exceptions, where multiple types are used to indicate the reason for the exception. However, for our purposes this model suffices — the results are easily extended to a richer model.

$$\begin{array}{lcl} \tau & ::= & \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid A \\ \xi & ::= & \bot_\Sigma \mid \top \end{array} \qquad \begin{array}{lcl} \omega & ::= & \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\} \\ \Sigma & ::= & \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \\ \Delta & ::= & \{A_1 \mapsto \omega_1, \ldots, A_n \mapsto \omega_n\} \end{array}$$

Subtyping We define two standard subtype relations: *width subtyping* $<:_w$ and *width-depth subtyping* $<:_d$. For brevity, we will use the term *depth subtyping* to refer to width-depth subtyping in the rest of this paper. Most of this paper is only concerned with width subtyping; thus, when not explicitly marked otherwise, $<:$ refers to width subtyping.

Width subtyping provides a uniform type view of the heap, see, for instance, invariant subtyping for `ML` references [Pie02], which is needed to support updates in the presence of aliases[2]; see below for a more thorough explanation.

Common to both width subtyping and depth subtyping are the rules for primitive types, exception types, and store types.

$$\tau <:_{w/d} \tau \quad \mathit{nat} <:_{w/d} \mathit{int}$$

$$\xi <:_{w/d} \xi \quad \top <:_{w/d} \bot_\Sigma \quad \frac{\Sigma_1 <:_{w/d} \Sigma_2}{\bot_{\Sigma_1} <:_{w/d} \bot_{\Sigma_2}}$$

$$\frac{\forall x \in \mathit{dom}(\Sigma_2).\ \Sigma_1(x) <:_{w/d} \Sigma_2(x)}{\Sigma_1 <:_{w/d} \Sigma_2}$$

[2] In the absence of additional analyses.

The difference between the relations is captured by the rules for subtyping of record types. Width subtyping allows records to be seen as smaller records while retaining the types of the remaining fields, i.e., bigger record types are subtypes of smaller given that all fields in the domain of the smaller record type are mapped to the same types in both the smaller and the bigger, while depth subtyping demands that all fields in the domain of the smaller are mapped to types that are depth subtypes of the corresponding types in the smaller record type.

$$\frac{\forall f \in dom(\omega_2).\ \omega_1.f = \omega_2.f}{\omega_1 <:_w \omega_2} \qquad \frac{\forall f \in dom(\omega_2).\ \omega_1.f <:_d \omega_2.f}{\omega_1 <:_d \omega_2}$$

The subtyping relations naturally induce a subtype relation on record type names. Thus, even though the system introduced so far is nominal, we use structural subtyping, defined on record identifiers as the smallest relation on record names $<:_{w/d}$ satisfying:

$$\frac{\Delta(A_1) <:_{w/d} \Delta(A_2)}{A_1 <:_{w/d} A_2}$$

This is in contrast to the more frequent use of pure nominal subtyping, where the programs explicitly declare what record names are subtypes of each other.

Width versus Depth Subtyping The subtyping relation defines when objects of one type can be safely seen as having another type. For instance, it is perfectly safe to view a natural number as an integer, since the set of integers include all natural numbers. It may seem natural to extend the subtyping relation to records based on the same subset argument; after all, the set of records with a field f holding a natural numbers is included in the set of records with the same field f holding an integer.

Such an extension of the subtyping relation to records is provided by depth-subtyping and is, in fact, perfectly sound in the presence of aliases as long as we only *read* from the records. However, in the presence of aliases and updates, depth subtyping is not sound, as illustrated by the following program where $\Delta(A) = \{f : nat\}$ and $\Delta(B) = \{f : int\}$.

```
A x := new A; B y := (B) x; y.f := -1; nat z := x.f;
```

In this example and the following examples we will use A, B, C, ... as record type names, and x, y, z, ... as variable names. In addition, for clarity, we will allow type annotations, type casts, and field assignment of constants — neither is necessary, but allows the examples to be more concise. For example, in the above program the cast in the assignment `y := (B) x` is needed to change the type of x to B before the assignment. Otherwise, the type of y would simply be overwritten by the type of x, i.e., A, since variable updates are flow-sensitive.

The example first creates a record with a field f of type natural numbers. Using depth subtyping we create an alias to the record with an integer field type. As noted above, reading the field via x and y is still sound - x has a more

$$\frac{\Sigma \vdash e : \tau, \xi}{\Sigma \vdash^{\dagger} x := e \Rightarrow \Sigma[x \mapsto \tau], \xi} \qquad \frac{\Sigma(x_1) = A \quad \Sigma(x_2) = \tau \quad \tau <: \Delta(A).f}{\Sigma \vdash^{\dagger} x_1.f := x_2 \Rightarrow \Sigma, \bot_{\Sigma}}$$

$$\frac{\Sigma_1 \vdash e : \mathit{bool}, \xi \quad \Sigma_1 \vdash^{\dagger} c_1 \Rightarrow \Sigma_2, \xi \quad \Sigma_1 \vdash^{\dagger} c_2 \Rightarrow \Sigma_2, \xi}{\Sigma_1 \vdash^{\dagger} \mathit{if}\ e\ c_1\ c_2 \Rightarrow \Sigma_2, \xi}$$

$$\frac{\Sigma \vdash e : \mathit{bool}, \xi \quad \Sigma \vdash^{\dagger} c \Rightarrow \Sigma, \xi}{\Sigma \vdash^{\dagger} \mathit{while}\ e\ c \Rightarrow \Sigma, \xi}$$

$$\frac{\Sigma_1 \vdash^{\dagger} c_1 \Rightarrow \Sigma_2, \xi \quad \Sigma_2 \vdash^{\dagger} c_2 \Rightarrow \Sigma_3, \xi}{\Sigma_1 \vdash^{\dagger} c_1; c_2 \Rightarrow \Sigma_3, \xi}$$

$$\frac{}{\Sigma \vdash^{\dagger} x := \mathit{new}(A) \Rightarrow \Sigma[x \mapsto A], \top}$$

$$\frac{}{\Sigma \vdash^{\dagger} \mathit{skip} \Rightarrow \Sigma, \top} \qquad \frac{\Sigma_2 \vdash^{\dagger} c \Rightarrow \Sigma_3, \xi_1 \quad \Sigma_1 <: \Sigma_2 \quad \Sigma_3 <: \Sigma_4 \quad \xi_1 <: \xi_2}{\Sigma_1 \vdash^{\dagger} c \Rightarrow \Sigma_4, \xi_2}$$

Figure 3: Type Rules for Commands

precise type, viewing the field as a natural number while y views it as an integer. However, the types permit us to update the field with an integer via y, and to read the written integer as a natural number via x, effectively introducing a cast going the opposite direction of the subtype hierarchy. Thus, a weakening rule as the one found in Figure 3 but based on depth subtyping rather than width subtyping is unsound.

It should be pointed out that writing is sound using a depth subtyping rule with the subtype of the fields inverted, i.e., we can view a record with an integer field as a record with a natural number field — all natural numbers are also integers and limiting the values that can be written into the field is unproblematic. For this reason reading is known as being *co-variant*, i.e., that sound subtyping with respect to reading extends structurally in the same way, and writing is known as being *contra-variant*, i.e., that sound subtyping with respect to writing extends structurally in the opposite way [Pie02]. In a system where the same type governs both reading and writing, the types of the fields must be both co-variant and contra-variant, i.e., they must be *invariant*. This is represented by the width subtyping that demands *equality* on the types of the fields, which implies invariance, since the subtyping relation is reflexive.

Expression Type Rules The typing judgment for expressions, $\Sigma \vdash e : \tau, \xi$, is read as the expression e is well-typed in the environment type Σ, with return type τ possibly resulting in exceptions as indicated by ξ. The type rules for the expressions are entirely standard, and omitted for brevity.

$$\frac{}{\delta \vdash b : \mathit{bool}} \qquad \frac{}{\delta \vdash i : \mathit{int}} \qquad \frac{i \geq 0}{\delta \vdash i : \mathit{nat}}$$

$$\frac{}{\delta \vdash \mathit{nil} : A} \qquad \frac{\delta(p) <: A}{\delta \vdash p : A}$$

$$\frac{\forall (f, \tau) \in \omega.\ \delta \vdash r.f : \tau}{\delta \vdash r : \omega} \qquad \frac{\forall (x, \tau) \in \Sigma.\ \delta \vdash s(x) : \tau}{\delta \vdash s : \Sigma}$$

$$\frac{\forall (p, A) \in \delta.\ \delta \vdash h(p) : \Delta(A)}{\delta \vdash h} \qquad \frac{\delta \vdash s : \Sigma \quad \delta \vdash h}{\delta \vdash (s, h) : \Sigma}$$

$$\frac{\delta \vdash v : \tau}{\delta \vdash v : \tau, \xi} \qquad \frac{}{\delta \vdash \bot : \tau, \bot_\Sigma}$$

$$\frac{\delta \vdash E : \Sigma_2}{\delta \vdash^\dagger \bot_E : \Sigma_1, \bot_{\Sigma_2}} \qquad \frac{\Sigma_1 \vdash^\dagger c \Rightarrow \Sigma_2, \xi \quad \delta \vdash E : \Sigma_1}{\delta \vdash^\dagger \langle E, c \rangle : \Sigma_2, \xi}$$

Figure 4: Well-formedness

Command Type Rules The type system for commands is flow sensitive; each command is typed with respect to a pre and a post environment type. The typing judgment for commands, $\Sigma_1 \vdash^{\mathbb{M}_I, \mathbb{M}_O, \mathcal{R}^+, \mathcal{R}^-} c \Rightarrow \Sigma_2, \xi$ is read as the command c is well-typed with respect to the abstract environment maps $\mathbb{M}_I$, and $\mathbb{M}_O$, the plugin $\mathcal{R}^+$, and the plugin $\mathcal{R}^-$ in the environment type Σ_1 resulting in the environment type Σ_2, possibly resulting in an exception as indicated by ξ. The standard type rules for commands are shown in Figure 3, where $\vdash^\dagger$ is used as short form for $\vdash^{\mathbb{M}_I, \mathbb{M}_O, \mathcal{R}^+, \mathcal{R}^-}$. The standard type system serves as the foundation for the extension with rules for structural weakening and strong updates.

4.1 Correctness

As is done by Pierce [Pie02] we split the correctness argument into two theorems, *progress* — intuitively, that well-typed commands and expressions are able to execute in all environments that conform to, i.e., are *well-formed* with respect to, the entry environment type of the command or expression — and *preservation* — intuitively, that the result of running the command or expression conforms to the exit environment type of the same. Together, progress and preservation guarantee proper execution of well-typed programs in all well-formed environments; progress and preservation repeatedly guarantee one step of execution, and that the result is well-formed.

Well-formedness We define two well-formedness relations, one corresponding to width subtypes, and one corresponding to depth subtypes. More precisely, well-formedness is formulated as a family of relations between values and types, indexed over pointer typings. The pointer typings, ranged over by δ, are maps from pointers to record type names and make the well-formedness relation induc-

tively definable also for cyclic heaps. The rules for well-formedness are found in Figure 4, where $\vdash^{\dagger}$ is used as short form for $\vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-}$. The well-formedness relations are entirely standard; the interaction between the well-formedness relation for stores and heaps, together with the well-formedness relation for records guarantees that if an environment is well-formed with respect to a store type and a pointer typing, then the pointer typing types at least all live (reachable) pointers.

Progress and Preservation of Expressions Preservation of expressions expresses that well-typed expressions preserve well-formedness under execution, i.e., for an expression e such that $\Sigma \vdash e : \tau, \xi$, running e in Σ-well-formed environments will result in τ, ξ-well-formed values.

Theorem 4.1 *Preservation of Types of Expressions*

$$\Sigma \vdash e : \tau, \xi \Longrightarrow \delta \vdash E : \Sigma \wedge \langle E, e\rangle \Downarrow v_{\perp} \Longrightarrow \delta \vdash v_{\perp} : \tau, \xi$$

Proof 4.1 *By induction over the derivation of* $\Sigma \vdash e : \tau, \xi$. *The proof is entirely standard and is omitted for brevity.*

Progress of expressions expresses that well-typed expressions are able to execute in correspondingly well-formed environments, i.e., for an expression e such that $\Sigma \vdash e : \tau, \xi$ it is possible to run e in any Σ-well-formed environment.

Theorem 4.2 *Progress of Expressions*

$$\Sigma \vdash e : \tau, \xi \Longrightarrow \delta \vdash E : \Sigma \Longrightarrow \exists v_{\perp}.\langle E, e\rangle \Downarrow v_{\perp}$$

Proof 4.2 *By induction over the derivation of* $\Sigma \vdash e : \tau, \xi$. *The proof is entirely standard and is omitted for brevity.*

Together progress and preservation for expressions guarantee that well-typed expressions are able to run in correspondingly well-formed environments, and that the results are well-formed.

Progress and Preservation of Commands Preservation of types of commands is formulated in essentially the same way as for expressions, i.e., for a command c such that $\Sigma_1 \vdash^{\dagger} c \Rightarrow \Sigma_2, \xi$, running c in any Σ_1-well-formed environment that is in any set of initial environments $\mathcal{C}$ which makes $\mathbb{M}_I$ an entry solution and $\mathbb{M}_O$ an exit solution for c will result in a Σ_2, ξ-well-formed configuration.

Theorem 4.3 *Preservation of Types of Commands*

$$\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2, \xi \wedge is_c^{\mathcal{C}}(\mathbb{M}_I) \wedge os_c^{\mathcal{C}}(\mathbb{M}_O) \Longrightarrow$$
$$E \in \mathcal{C} \wedge \delta_1 \vdash E : \Sigma_1 \wedge \langle E, c\rangle \rightarrow C \Longrightarrow \exists \delta_2.\ \delta_2 \vdash^{\dagger} C : \Sigma_2, \xi$$

Proof 4.3 *By induction on the derivation of* $\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2, \xi$. *The proof is entirely standard since the standard type rules do not make use of the parameterized information and is omitted for brevity.*

Progress of commands expresses that well-typed commands are able to run in correspondingly well-formed environments, i.e., for a command c such that $\Sigma_1 \vdash^{\dagger} c \Rightarrow \Sigma_2, \xi$, it is possible to run c in any Σ_1-well-formed environment.

Theorem 4.4 *Progress of Commands*

$$\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2, \xi \wedge is_c^{\mathcal{C}}(\mathbb{M}_I) \wedge os_c^{\mathcal{C}}(\mathbb{M}_O) \Longrightarrow E \in \mathcal{C} \wedge \delta \vdash E : \Sigma_1 \Longrightarrow \exists C.\ \langle E, c\rangle \rightarrow C$$

Proof 4.4 *By induction on the derivation of* $\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2, \xi$. *The proof is entirely standard since the standard type rules do not make use of the parameterized information and is omitted for brevity.*

As above progress and preservation interact to guarantee successful execution of well-typed commands in well-formed environments. Let $\langle E, c\rangle \rightarrow^n C$ be the obvious lifting of the small step evaluation to evaluation of n consecutive steps. We formulate the top-level correctness of commands, that well-typed commands terminate in well-formed environments or result in well-formed configurations regardless of the number of execution steps, in the following way, where T ranges over terminal configurations.

Theorem 4.5 *Top-level Correctness of Commands*

$$\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c_1 \Rightarrow \Sigma_2, \xi \wedge is_{c_1}^{\mathcal{C}}(\mathbb{M}_I) \wedge os_{c_1}^{\mathcal{C}}(\mathbb{M}_O) \Longrightarrow E_1 \in \mathcal{C} \wedge \delta_1 \vdash E_1 : \Sigma_1 \wedge \langle E_1, c_1\rangle \rightarrow^n C \Longrightarrow \exists \delta_2\ .\ \delta_2 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} C : \Sigma_2, \xi$$

Proof 4.5 *The proof of top-level correctness for commands proceeds by induction over the number of execution steps. The proof is completely independent on the parameterized information, and thus valid for all possible parameterizations and proceed by induction over the number of execution steps. For brevity we refer to [GH08] for the details of the proof.*

5 Heap Types and Aliases

The type system presented in Section 4 allows the types of the variables to change; after an assignment to a variable, the type of the variable becomes the type of the expression that was assigned to the variable. This is an example of a *flow-sensitive* type system. As discussed, this type scheme does not immediately extend to records. Instead, it is common to demand type invariance for heap locations to guarantee a uniform type view of the heap.

Let sl range over symbolic locations, defined as follows.

$$sl ::= x \mid sl.f$$

The meaning of the symbolic locations with respect to a concrete environment is defined by recursive dereference. In a given environment, each symbolic location refers at most one concrete location, but different symbolic locations may refer to the same concrete location. When this occurs, we say that the symbolic locations are aliased.

This section explores the details of the interaction between aliases and subtyping in the presence of reading and writing, and how alias information can be obtained using the plugin framework. The following two sections show how this information can be used to achieve a certain degree of the freedom enjoyed by variable types — simplified, the alias information is used to make sure that all aliased locations agree on the type.

5.1 Flow-sensitive Heap Types

Consider the flow-sensitive type rule for variable assignment from Section 4.

$$\frac{\Sigma \vdash e : \tau, \xi}{\Sigma \vdash^{\dagger} x := e \Rightarrow \Sigma[x \mapsto \tau], \xi}$$

The soundness of this rule relies on the fact that each variable is stored at a unique place in the store, which is not shared by any other variables. Thus, writing to a variable does not modify the value, and hence neither the type, of any of the other variables.

A similar rule for records on the heap is not immediately possible in the presence of aliasing. Similar to above, if such a rule is provided we have a direct way of making a single concrete location seen as having two different types. As shown above, such a situation is equivalent to having a way of freely casting between the two types by writing into the concrete location via one symbolic location, and reading from the other symbolic location, as illustrated by the following program assuming that $\Delta(A) = \{f : int\}$.

```
A x := new A;  A y := x; y.f := true; int z := x.f;
```

First, x and y are initialized to point to the same record. Thereafter, the field of that record is updated with a boolean via y, causing the type of y to become $\{f : bool\}$. Integers written via x can now be read as booleans via y, and vice versa.

5.2 Alias Information

Alias information is information about which symbolic locations may be or are guaranteed to be aliased with each other. There are two forms of aliases, *may*- and *must*-aliases, corresponding to whether two symbolic locations may be aliased, i.e., it cannot be ruled out that they are aliased, or must be aliased,

i.e., they are always aliased. Intuitively, if x and y are may-aliased then it may be the case that in one of the program runs x and y contain the same pointer. On the other hand if x and y are must-aliased then it must be the case that they contain the same pointer in every program run.

There are two common forms of alias information: as a map from program points to 1) relations on symbolic locations, or 2) structural environments — see [Deu94] for references of both methods.

In the former, if a pair of symbolic locations (sl_1, sl_2) are related for some program point labeled with l then, in the may-alias case, it cannot be excluded that sl_1 and sl_2 contain the same concrete pointer at l in some program run, and, in the must-alias case, it must be the case that sl_1 and sl_2 contain the same pointer at l in all program runs. To keep the alias information finite in the presence of cycles, two important properties of (both may- and must-) alias information are used. The first property is a form of substitutivity property

$$(sl_1, sl_2) \wedge (sl_3, sl_4) \Longrightarrow (sl_3[sl_1/sl_2], sl_4[sl_1/sl_2]) \tag{1}$$

which expresses that if sl_1 and sl_2 are aliased then one can form new aliases by replacing sl_1 with sl_2 in other aliases. For example, if (x, y) and $(x, x.f)$, then we know from this rule that $(y, y.f)$. This property subsumes transitivity, since if (sl_1, sl_2) and (sl_2, sl_3) then (sl_1, sl_3). The second property

$$(sl_1, sl_2) \Longrightarrow (sl_1.f, sl_2.f) \tag{2}$$

expresses that anything reachable from an alias is also an alias.

For our purposes the second form of alias information — the structural environments — is more convenient to work with. The idea behind structural environments is to have an abstract representation that captures information about the common structure of a set of concrete environments. This is achieved by using abstract structural pointers with the property that (may- and must-) aliased symbolic locations contain the same structural pointer.

Syntax of Structural Environments The syntax of the structural environments follows the syntax of the values, with pointers represented by abstract pointers, and all other values represented by an abstract dummy.

$$\begin{array}{lll} \widehat{v} & ::= & \widehat{p} \mid \bullet \\ \widehat{E} & ::= & (\widehat{s}, \widehat{h}) \end{array} \qquad \begin{array}{lll} \widehat{r} & ::= & \{f_1 \mapsto \widehat{v}_1, \ldots, f_n \mapsto \widehat{v}_n\} \\ \widehat{s} & ::= & \{x_1 \mapsto \widehat{v}_1, \ldots, x_n \mapsto \widehat{v}_n\} \\ \widehat{h} & ::= & \{\widehat{p}_1 \mapsto \widehat{r}_1, \ldots \widehat{p}_n \mapsto \widehat{r}_n\} \end{array}$$

That is, the structural values $\widehat{v}$ are either structural pointers, ranged over by $\widehat{p}$, or any other value represented by $\bullet$. A structural record $\widehat{r}$ is a map from field names to structural values, a structural store $\widehat{s}$ is a map from variable names to structural values, and a structural heap $\widehat{h}$ is a map from structural pointers to structural records. Finally, the structural environments $\widehat{E}$ are pairs of structural stores and heaps.

May-alias interpretation We define the may-alias meaning of the structural heaps in terms of a may-alias concretization function γ^+. In the following we will drop the superscript when possible without risk of confusion. Let ξ range over maps from structural pointers to non-empty sets of concrete pointers. To make sure that different structural pointers get mapped to different concrete pointers we parameterize the concretization function over a structural pointer valuation with pairwise disjoint codomain, excluding the null-pointer. We say that such pointer valuations are *may-alias sound*.

Definition 5.1 (May-alias sound pointer valuation) *A pointer valuation ξ is may-alias sound if it does not map anything to the null-pointer and has a pairwise disjoint codomain, i.e.*

$$\forall \widehat{p}_1, \widehat{p}_2 \in dom(\xi) \;.\; \widehat{p}_1 \neq \widehat{p}_2 \Longrightarrow \xi(\widehat{p}_1) \cap \xi(\widehat{p}_2) = \emptyset$$

Let ξ^+ range over the set of may-alias sound pointer valuations.

Let $\mathcal{V}_{\backslash p}$ be the set of concrete values excluding the pointers. The concretization for primitive values is defined as follows.

$$\gamma^+_{\xi^+}(\bullet) = \mathcal{V}_{\backslash p} \qquad \gamma^+_{\xi^+}(\widehat{p}) = \xi^+(\widehat{p}) \cup \{nil\}$$

The concretization function is extended structurally while allowing all possible combinations of concrete pointers as defined by the pointer valuation. For records we let $\gamma^+_{\xi^+}(\widehat{r}) = \{r \mid f \in dom(\widehat{r}), r.f \in \gamma^+_{\xi^+}(\widehat{r}.f)\}$, and similarly for stores $\gamma^+_{\xi^+}(\widehat{s}) = \{s \mid x \in dom(\widehat{s}), s(x) \in \gamma^+_{\xi^+}(\widehat{s}(x))\}$. For heaps we let $\gamma^+_{\xi^+}(\widehat{h}) = \{h \mid \widehat{p} \in dom(\widehat{h}), p \in \xi^+(\widehat{p}), h(p) \in \gamma^+_{\xi^+}(\widehat{h}(\widehat{p}))\}$. Finally, we define $\gamma^+_{\xi^+}(\widehat{E})$ for environments by combining the results from the concretization functions for heaps and stores using the same pointer valuation.

The set of concrete environments associated with one particular structural environment is the union over all may-alias sound pointer valuations.

$$\gamma^+(\widehat{E}) = \bigcup_{\xi^+} \{\gamma^+_{\xi^+}(\widehat{E})\}$$

Must-alias interpretation The concretization function above defines the meaning of may-aliases; with a small change in the interpretation of the structural values we can define the meaning of must-aliases. Let γ^- denote the concretization function for must-aliases. As above when there is no risk of confusion the superscript is dropped. A pointer valuation is *must-alias sound* if all sets in its codomain are singleton.

Definition 5.2 (Must-alias sound pointer valuation) *A pointer valuation is must-alias sound if all sets in its codomain are singleton and do not contain the null-pointer, i.e.*

$$\forall \widehat{p} \in dom(\xi) \;.\; |\xi(\widehat{p})| = 1$$

Let ξ^- range over must-alias sound pointer valuations.

For structural values we define the must-alias concretization function as

$$\gamma^-_{\xi^-}(\bullet) = \mathcal{V} \qquad \gamma^-_{\xi^-}(\widehat{p}) = \xi^-(\widehat{p})$$

with the difference that, unlike $\mathcal{V}_{\setminus p}$, $\mathcal{V}$ ranges over the set of all concrete values including the pointers. The structural extension of the concretization function to structural environments is done in exactly the same way as above.

As before, the set of concrete environments associated with one particular structural environment is the union over all valid pointer valuations.

$$\gamma^-(\widehat{E}) = \bigcup_{\xi^-} \{\gamma^-_{\xi^-}(\widehat{E})\}$$

This means that the must-alias information only conveys information about which pointer locations must be equal — it does not rule out any other aliases, and considers all locations to be possibly aliased with any other location.

5.3 Plugins are Under Approximations

Sound may-alias information can be seen as an over approximation of the possibly *aliased* locations, i.e., it is safe to consider more locations to be aliased than actually are.

When designing plugin properties for probing may-alias information we must take into consideration that plugins are by definition *under approximations*, and as such not suitable for probing over approximations — if two locations are not marked as being aliases the conclusion is that they are unaliased, however a plugin may by definition freely exclude locations from the relation, making this conclusion invalid.

The solution to this is to use the dual interpretation of may-aliases — must-not aliases — i.e., the under approximation of guaranteed *unaliased* locations. Thus, instead of using equality on pointers as the relation for our plugin property we use inequality.

5.4 Extracting May-Aliases

Using the may-alias plugin $\mathcal{R}^+$ defined above we can extract may-alias information by a traversal rooted in the variables of pointer type.

The algorithm takes a may-alias plugin `R`, an abstract environment `aenv`, a list of previously visited symbolic locations `vs`, a structural environment which is modified during the traversal `env`, and a work list of symbolic locations yet to be visited `ls`. Each iteration removes the topmost symbolic location of the work list, and checks it against all previously visited symbolic locations. If it is guaranteed to be unaliased with all previous locations a fresh structural pointer is introduced and the symbolic location associated with it; otherwise, the symbolic location is associated with the structural pointer of the found previous location. Whenever a new symbolic location is introduced all relevant

succeeding symbolic locations (the fields of pointer type) are added at the end of the work list.

In the pseudo code below, `lookup env v` gets the structural pointer associated with `v` in the structural environment `env`, `env[p -> *]` returns an updated version of `env` where `p` is associated with an initialization record that is type compatible with the location associated with `p`, i.e., the fields of pointer type are empty (they will be updated by subsequent iterations) and all other fields contain •, `env[l -> p]` returns an updated version of `env` where the location `l` is associated with the structural pointer `p`, and **fields of** `l` returns the list of symbolic locations of the fields of `l` of pointer type.[3]

```
extract _ _ _  env [] = env
extract aenv R vs env (l:ls) =
 case find (not . R aenv l) vs of
  None -> p    = fresh pointer
          env' = env[p -> *][l -> p]
          lfs  = fields of l
          extract aenv R (l:vs) env' (ls++lfs)

  Some v -> p = lookup env v
            env' = env[l -> p]
            extract aenv R vs env' ls
```

We define the extraction function η^+ for may-aliases in terms of the above *extract* function as follows, where $init_\Gamma$ is the list of initial symbolic locations — the variables of pointer type (as given by the store type).

$$\eta^+(\mathbb{E}, \mathcal{R}^+, \Gamma) = extract\ \mathbb{E}\ \mathcal{R}^+\ [\,]\ ([\,], [\,])\ init_\Gamma$$

Thus, $\eta^+(\mathbb{E}, \mathcal{R}^+, \Gamma)$ extracts a may-alias view of $\mathbb{E}$, using $\mathcal{R}^+$ starting in the variables given pointer types by Γ.

The correctness of the algorithm relies on the substitutivity property of the may-alias information. Termination of the algorithm relies on the abstract environment and the number of fields being finite and every non-terminated path of field references in the may-alias information containing a cycle, i.e., there must not be an infinite number of unaliased locations; a formal definition of this cyclic path property is found in Appendix A. For each underlying may-alias information there exist plugins for which these properties hold.

5.5 Extracting Must-Aliases

Similar to above, we can extract must-alias information using the must-alias plugin $\mathcal{R}^-$.

The algorithm takes a must-alias plugin `R`, an abstract environment `aenv`, a list of previously visited symbolic locations `vs`, a a structural environment which is modified during the traversal `env`, and a work list of symbolic locations yet to be visited `ls`. Each iteration removes the topmost symbolic location of

[3] All aliased locations are required to have the same type.

the work list, and checks it against all previously visited symbolic locations. If it is must-aliased with any of the previously inspected symbolic locations it is associated with the structural pointer of the found previous location; otherwise, it is checked whether it is must-aliased with itself, which would imply that it is guaranteed not to be a null-pointer.[4] If it is must-aliased with itself a fresh structural pointer is introduced and the symbolic location associated with it. If it is not then the symbolic location is associated with •.

```
extract _ _ _  env [] = env
extract aenv R vs env (l:ls) =
 case find (not . R aenv l) vs of
  None -> if R aenv l l then
            p    = fresh pointer
            env' = env[p -> *][l -> p]
            lfs  = fields of l
            extract aenv R (l:vs) env' (ls++lfs)
          else
            env' = env[l -> •]
            extract aenv R vs env' ls

  Some v -> p    = lookup env v
            env' = env[l -> p]
            extract aenv R vs env' ls
```

The extraction function η^- for must-aliases is defined in the same way as the extraction function for may-aliases and correctness and termination of the algorithm rely on the same properties.

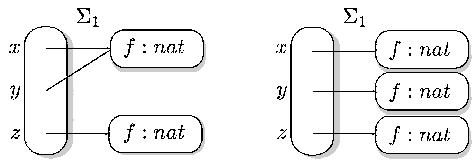
6 Structural Weakening

This section details how may-alias information can be used to safely weaken the types of heap locations. The section begins with two examples that show why a naive extension of the standard weakening rule is unsound, and how structural alias information can be used to provide a sound weakening rule by demanding that all aliased locations are subject to the same type changes. Thereafter, we introduce the basis for the weakening, the decorating structural well-formedness — essentially a well-formedness relation for structural values — and show how it can be used to create a sound weakening rule. The section ends with a small example illustrating the use of the weakening rule.

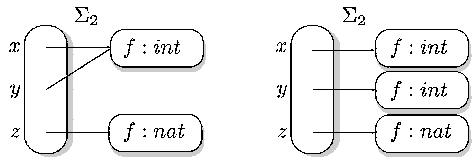
As we saw in Section 4, a weakening rule based on depth-subtyping is not sound in the presence of aliases and updates. The problem is that depth-subtyping makes it possible to create different type views of aliased symbolic locations. Without alias information we are forced to impose an invariant type view of all locations that may be aliased using width-subtyping; with may-alias information it is possible to relax this demand and demand an invariant type view only on the locations that are may-aliased.

[4] This is the case since the plugin for must-alias excludes the null-pointer.

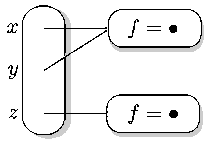
To illustrate this, assume a set of concrete environments S, e.g., the set of environments reaching a certain program point in a particular program. Assume that x and y are may-aliased and that x and y are unaliased with z, i.e., there exist at least one environment in S, where x and y point to the same record, and in no environments in S does z point to the same record as x or y. Assume further that all of x, y and z point to records of one field f holding a natural number, i.e., that the environments in S are well-formed in $\Sigma_1 = \{x : A, y : A, z : A\}$, where $\Delta(A) = \{f : nat\}$. The possible heaps in S (up-to type constrained isomorphism and omitting null-pointers) can be illustrated as follows.



It is clear that we can safely change the types of x and y to hold an integer as long as we change both, i.e., all environments in S are well-formed in $\Sigma_2 = \{x : B, y : B, z : A\}$, where $\Delta(B) = \{f : int\}$.



To see how structural may-alias information can be used to achieve this consider the structural representation of the situation above: $\widehat{s} = \{x \mapsto \widehat{p}_1, y \mapsto \widehat{p}_1, z \mapsto \widehat{p}_2\}$, with $\widehat{h} = \{\widehat{p}_1 \mapsto \{f \mapsto \bullet\}, \widehat{p}_2 \mapsto \{f \mapsto \bullet\}\}$.



As can be seen in the picture and as was described in Section 5.2 all locations that may be aliased contain the same structural pointer. Thus, the same ideas underlying width well-formedness for concrete environments can be used for structural may-alias information to ensure a uniform type-view for all may-aliased locations.

Decorated Structural May-Aliases We define the well-formedness relation for structural may-aliases following the standard well-formedness; in addition we let the structural well-formedness produce a type decorated version of the structural environment — the use of the decoration will become apparent below.

$$\Omega \vdash \bullet : \tau \curvearrowright \bullet_\tau,\ \tau \in \{nat, int, bool\} \qquad \frac{\Omega(\widehat{p}) <: A \qquad A <: \Omega(\widehat{p})}{\Omega \vdash \widehat{p} : A \curvearrowright \widehat{p}}$$

$$\frac{(f,\tau) \in \omega.\ \Omega \vdash \widehat{r}.f : \tau \curvearrowright \widehat{r}_d.f \quad dom(\widehat{r}) = dom(\widehat{r}_d) = dom(\omega)}{\Omega \vdash \widehat{r} : \omega \curvearrowright \widehat{r}_d}$$

$$\frac{\forall (x,\tau) \in \Sigma.\ \Omega \vdash \widehat{s}(x) : \tau \curvearrowright \widehat{s}_d(x) \quad dom(\widehat{s}) = dom(\widehat{s}_d) = dom(\Sigma)}{\Omega \vdash \widehat{s} : \Sigma \curvearrowright \widehat{s}_d}$$

$$\frac{\forall (\widehat{p}, A) \in \Omega.\ \Omega \vdash \widehat{h}(\widehat{p}) : \Delta(A) \curvearrowright \widehat{h}_d(\widehat{p}) \quad dom(\widehat{h}) = dom(\widehat{h}_d) = dom(\Omega)}{\Omega \vdash \widehat{h} \curvearrowright \widehat{h}_d}$$

$$\frac{\Omega \vdash \widehat{s} : \Sigma \curvearrowright \widehat{s}_d \qquad \Omega \vdash \widehat{h} \curvearrowright \widehat{h}_d}{\Omega \vdash (\widehat{s}, \widehat{h}) : \Sigma \curvearrowright (\widehat{s}_d, \widehat{h}_d)}$$

Figure 5: Decorating Structural Well-formedness

The language for the decorated structural may-aliases is identical to the language for the structural may-aliases with the addition of a type decoration on all occurrences of $\bullet$:

$$\widehat{v} \quad ::= \quad \widehat{p} \mid \bullet_\tau$$

When needed we use $\widehat{v}_d$, $\widehat{r}_d$, $\widehat{s}_d$, $\widehat{h}_d$, and $\widehat{E}_d$ to distinguish decorated values, records, stores, heaps and environments from the undecorated structural may-alias counterparts.

Concretization of Decorated Structural May-Aliases The concretization is constrained to the meaning of the type annotation, instead of all values apart from the pointers

$$\gamma^+_{\xi^+}(\bullet_\tau) = [\![\tau]\!]$$

where $[\![nat]\!]$ is the set of natural numbers, $[\![int]\!]$ the set of integers, and $[\![bool]\!]$ the set of booleans.

Decorating Structural Well-formedness Let Ω range over structural pointer typings, i.e., maps from structural pointers to record identifiers. The rules for the *decorating structural well-formedness* are found in Figure 5, and are easily extended to decorated structural values by replacing the rule for $\bullet$ with:

$$\frac{\tau_1 <: \tau_2}{\Omega \vdash \bullet_{\tau_1} : \tau_2 \curvearrowright \bullet_{\tau_1}} \quad \tau_1, \tau_2 \in \{nat, int, bool\}$$

The decorating structural well-formedness has two important properties. First, the type decoration does not exclude any well-formed environments.

Lemma 6.1 (Stability of Type Decoration)

$$\Omega_{max} \vdash \widehat{E} : \Sigma \curvearrowright \widehat{E}_d \wedge \delta \vdash E : \Sigma \wedge E \in \gamma^+(\widehat{E}) \Longrightarrow E \in \gamma^+(\widehat{E}_d)$$

where Ω_{max} denotes the maximal Ω *with respect to which $\widehat{E}$ is well-formed in Σ. See Appendix B for details.*

Proof 6.1 *First, $E \in \gamma^+(\widehat{E})$ implies the existence of ξ^+ such that $E \in \gamma^+_{\xi^+}(\widehat{E})$. Now, Lemma B.2 gives us that $E \in \gamma^+_{\xi^+_{E,\Sigma}}(\widehat{E})$. The result is immediate from Lemma B.7, and Lemma B.9.*

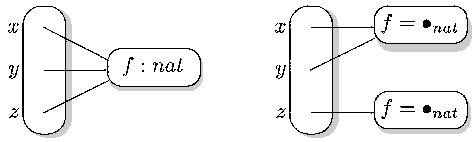
Second, well-formedness is preserved by concretization of the *decorated* structural environment. Let δ_{Ω,ξ^+} be the pointer typing induced by Ω and ξ^+, i.e., $\delta_{\Omega,\xi^+}(p) = \Omega(\xi^{+-1}(p))$. We know that ξ^{+-1} exists, since all may-alias sound pointer valuations are injective.

Lemma 6.2 (Preservation of Well-formedness under Concretization)

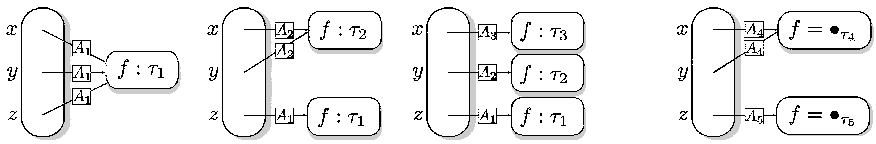
$$\Omega \vdash \widehat{E}_d : \Sigma \wedge E \in \gamma^+_{\xi^+}(\widehat{E}_d) \Longrightarrow \delta_{\Omega,\xi^+} \vdash E : \Sigma$$

Proof 6.2 *Given that $E = (s, h)$ we must show that $\delta_{\Omega,\xi^+} \vdash s : \Sigma$, and that $\delta_{\Omega,\xi^+} \vdash h$. The result is immediate from Lemma C.3, and Lemma C.5.*

One way of viewing the decorated structural may-alias representation is as a may-alias aware environment type, i.e., an environment type where types have been specialized to the may-alias structure. The following picture of the standard type view to the left and the decorated structured may-alias environment from the example above to the right illustrates this idea.



It is easy to see how the structural may-alias information limits the freedom of the types as illustrated by the following picture showing three different environment types and a structural environment (rightmost). The structural environment gives a limit to the maximal possible type structure in the sense that it defines which symbolic locations must have the same types. Thus, intuitively, the two leftmost environment types are compatible, but not the third, since the third tries to give x and y different types — A_3 and A_2 respectively.



For a more detailed explanation, assume that A_1, A_2, and A_3 are different. The first example can be accommodated by choosing $A_4 = A_5 = A_1$ and thus $\tau_4 = \tau_5 = \tau_1$, the second example by choosing $A_4 = A_2$ and $A_5 = A_1$, and thus $\tau_4 = \tau_2$ and $\tau_5 = \tau_1$, whereas the third example cannot be accommodated, since that would imply choosing A_4 and τ_4 to two different types corresponding to different type views for aliased locations.

With this view the well-formedness for decorated structural values can be used to formulate something that can be seen as a constrained depth-subtype relation. To see this let $\Omega_1 \vdash \widehat{E} : \Sigma_1 \curvearrowright \widehat{E}_d \wedge \Omega_2 \vdash \widehat{E}_d : \Sigma_2$ be written $\Sigma_1 <:_{\widehat{E}} \Sigma_2$. We have the following semantic property.

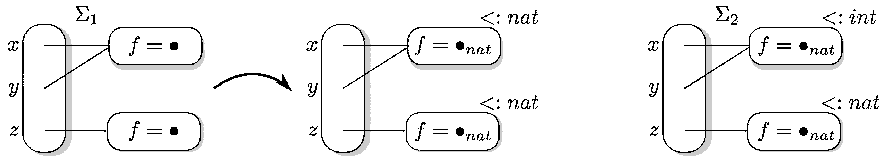
Lemma 6.3 (Structural Subtype)

$$\Sigma_1 <:_{\widehat{E}} \Sigma_2 \wedge E \in \gamma^+_{\xi^+}(\widehat{E}) \wedge \delta \vdash E : \Sigma_1 \Longrightarrow \exists \delta.\ \delta \vdash E : \Sigma_2$$

Proof 6.3 *We have that* (1) $\Omega_1 \vdash \widehat{E} : \Sigma_1 \curvearrowright \widehat{E}_d$, (2) $\Omega_2 \vdash \widehat{E}_d : \Sigma_2$, (3) $E \in \gamma^+_{\xi^+}(\widehat{E})$, *and* (4) $\delta \vdash E : \Sigma_1$

First, Lemma 6.1 together with $(1, 4, 3)$ *gives us that* (5) $E \in \gamma^+_{\xi^+}(\widehat{E}_d)$. *Now, Lemma 6.2 together with* $(2, 5)$ *gives us that* $\delta_{\Omega_2,\xi^+} \vdash E : \Sigma$ *and we are done.*

To illustrate the use consider the example from above where $\widehat{s} = \{x \mapsto \widehat{p}_1, y \mapsto \widehat{p}_1, z \mapsto \widehat{p}_2\}$, with $\widehat{h} = \{\widehat{p}_1 \mapsto \{f \mapsto \bullet\}, \widehat{p}_2 \mapsto \{f \mapsto \bullet\}\}$, $\Sigma_1 = \{x : A, y : A, z : A\}$ and $\Sigma_2 = \{x : B, y : B, z : A\}$, where $\Delta(A) = \{f : nat\}$, and $\Delta(B) = \{f : int\}$. In this example the two steps of $\Sigma_1 <:_{(\widehat{s},\widehat{h})} \Sigma_2$ are $\{\widehat{p}_1 \mapsto A, \widehat{p}_2 \mapsto A\} \vdash (\widehat{s}, \widehat{h}) : \Sigma_1 \curvearrowright (\widehat{s}_d, \widehat{h}_d)$, and $\{\widehat{p}_1 \mapsto B, \widehat{p}_2 \mapsto A\} \vdash (\widehat{s}_d, \widehat{h}_d) : \Sigma_2$, for $\widehat{s}_d = \widehat{s}$ and $\widehat{h}_d = \{\widehat{p}_1 \mapsto \{f \mapsto \bullet_{nat}\}, \widehat{p}_2 \mapsto \{f \mapsto \bullet_{nat}\}\}$, since $nat <: int$, as illustrated by the following picture, where the subtyping annotations express the demands put on the structural environments by Σ_1 in the middle structural environment and Σ_2 in the rightmost structural environment.



The picture illustrates how Σ_1 has decorated the the original structural environment $(\widehat{s}, \widehat{h})$, and how Σ_2 is able to change the type view of x and y to a super type of the previous type recorded by the decorated structural environment.

Structural Weakening Based on this we can create a new weakening rule based on well-formedness where the structural representations of the entry and exit environments of a command c are used to ensure that all aliased pointers have compatible type views. Let $\mathbb{M}_I$ and $\mathbb{M}_O$ range over entry and exit solutions,

respectively, and let η^+ be the may-alias extraction function.

$$\frac{\Sigma_2 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_3, \xi_1 \quad \xi_1 <: \xi_2 \quad \Sigma_1 <:_{\eta^+(\mathbb{M}_I(l_1),\mathcal{R}^+,\Sigma_1)} \Sigma_2 \quad \Sigma_3 <:_{\eta^+(\mathbb{M}_O(l_2),\mathcal{R}^+,\Sigma_3)} \Sigma_4}{\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} (c)^{l_1}_{l_2} \Rightarrow \Sigma_4, \xi_2}$$

We prove soundness of the structural weakening rule by proving preservation of types for it.

Lemma 6.4 (Preservation of Types of Structural Weakening)

$$\begin{aligned}\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2, \xi \wedge is_c^{\mathcal{C}}(\mathbb{M}_I) \wedge os_c^{\mathcal{C}}(\mathbb{M}_O) \Longrightarrow \\ E \in \mathcal{C} \wedge \delta_1 \vdash E : \Sigma_1 \wedge \langle E, c\rangle \rightarrow C \Longrightarrow \\ \exists \delta_2.\ \delta_2 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} C : \Sigma_2, \xi\end{aligned}$$

Proof 6.4 *Assume* (1) $\Sigma_1 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2, \xi$, (2) $is_c^{\mathcal{C}}(\mathbb{M}_I)$, (3) $os_c^{\mathcal{C}}(\mathbb{M}_O)$, (4) $E \in \mathcal{C}$, (5) $\delta_1 \vdash E : \Sigma_1$ *and* (6) $\langle E, c\rangle \rightarrow C$.

(1) *gives* (7) $\Sigma_1' \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c \Rightarrow \Sigma_2', \xi'$, (8) $\xi' <: \xi$, (9) $\Sigma_1 <:_{\eta^+(\mathbb{M}_I(l_1),\mathcal{R}^+,\Sigma_1)} \Sigma_1'$, *and* (10) $\Sigma_2' <:_{\eta^+(\mathbb{M}_O(l_2),\mathcal{R}^+,\Sigma_2')} \Sigma_2$.

(2) *and* (4) *gives* (13)$E \in \gamma^+(\mathbb{M}_I(l_1))$ *which together with soundness of the extraction function for may-aliases gives* (14)$E \in \gamma^+_{\xi_1^+}(\eta^+(\mathbb{M}_I(l_1), \mathcal{R}^+, \Sigma_1))$ *for some* ξ_1^+. *From this Lemma 6.3* (11)$\delta_1' \vdash E : \Sigma_1'$ *for some* δ_1'. *Now the induction hypothesis is applicable, which gives* (12) $\delta_2 \vdash C : \Sigma_2', \xi$ *for some* δ_2. *We proceed with an analysis of* (12).

abnormal termination *This case gives* $\delta_2 \vdash E_2 : \Sigma_e$ *for some* Σ_3, *where* $C = \bot_{E_3}$, *which immediately gives* $\delta_2 \vdash \bot_{E_2} : \Sigma_2', \bot_{\Sigma_3}$.

termination *This case gives* $\delta_2 \vdash E_2 : \Sigma_2'$, *where* $C = E_2$. *From* $os_c^{\mathcal{C}}(\mathbb{M}_O)$, *we get that* $E_2 \in \gamma^+(\mathbb{M}_O(l_2))$, *and, thus, from the soundness of the extraction function that* $E_2 \in \gamma^+_{\xi_2^+}(\eta^+(\mathbb{M}_O(l_2), \mathcal{R}^+, \Sigma_2'))$ *for some* ξ_2^+. *Similar to above Lemma 6.3 gives* $\delta_2' \vdash E_2 : \Sigma_2$, *for some* δ_2' *which gives us that* $\delta_2' \vdash E_2 : \Sigma_2, \xi$ *from which the result is immediate.*

non-termination *This case gives* $\Sigma_3 \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c' \Rightarrow \Sigma_2'$ *for some* Σ_3, *and* $\delta_2 \vdash E : \Sigma_3$. *In the same way as above we establish that* $\delta_2' \vdash E_2 : \Sigma_2$ *for some* δ_2'. *It now remains to show that* $\Sigma' \vdash^{\mathbb{M}_I,\mathbb{M}_O,\mathcal{R}^+,\mathcal{R}^-} c' \Rightarrow \Sigma_2$, *which is immediate from the structural weakening rule.*

Example Use We end this section with a small example of the use of structural weakening for achieving a limited form of flow-sensitive types on the heap. Assuming $\Delta(A) = \{f : nat\}$ consider the following program, which is not typable in the standard type system since *int* is not a subtype of *nat*.

```
A x = new A; A y := x; x.f := 0; x.f := -1;
```

However, even a simplistic alias analysis is able to determine that x and y are may-aliased, and unaliased with all other locations. This means that before $x.f := -1$ the type of x and y can be weakened to B, given that $\Delta(B) = \{f : \mathit{int}\}$, which allows us to perform the update. Since all aliases have their types changed uniformly, the pitfall of introducing the possibility of casting values is avoided.

7 Strong Updates

This section shows how may-alias information and must-alias information can be combined to allow for heap updates that do not follow the subtype hierarchy — similar to the updates of variables where the variable type environment is updated with the type of the value written into the variable. This differs from the structural weakening of the previous section, where the structure of the environments was used to express which types soundly described the environments, and the update was supported by finding a type in which the update was supported. For strong updates the actual update is more central; the old environment is typically not well-formed in the new environment type, and vice versa. Consider the following tiny program, which is typable in pre-type $\Sigma_1 = \{x : \tau\}$ for any type τ, and post-type $\Sigma_2 = \{x : \mathit{bool}\}$ using the flow sensitive type rule for variables.

```
x := 0; x := true;
```

Clearly, after assigning a boolean to x the environment is not typable in the post-type of $x := 0$, in which the type of x is nat.

The soundness of the flow-sensitive type rule for variables comes from the fact that no variables are aliased. In the same way, if a structural (may-) pointer is uniquely associated with a symbolic location we know that that symbolic location is alias free and can safely be strongly updated. However, demanding that a location is completely unaliased to support strong updates is unnecessarily restrictive. For instance, if we know that all aliases to the symbolic location are must-aliases, we know that a strong update is safe, given that we change the type of all must-aliases accordingly. Thus, it would be natural to expect the following program to be typable in the empty pre-type $\{\ \}$ and post-type $\{x : B, y : B\}$, where $\Delta(A) = \{f : \tau\}$ for some τ, and $\Delta(B) = \{f : \mathit{bool}\}$.

```
A x := new A; A y := x; x.f := true;
```

The previous section showed how may-alias information can be used to support weakening, and how weakening can be used to support a limited form of flow sensitive types. This was achieved by the use of a structural width well-formedness relation that guaranteed concrete width well-formedness. To fit with the results of the previous section, and with the correctness proof of the standard type system we will use preservation of width well-formedness as the base for the correctness argument of this section. This restricts the result to updates of must-aliased location *that are not reachable via may-aliases*. This restriction is

justified by the fact that the presence of may-aliases would constrain the update to follow the sub-type hierarchy, which together with the demand of concrete width well-formedness would result in the same expressive power as structural weakening.

Combined May- and Must-alias Information The approach we consider is based on a sound merge of may- and must-alias information that guarantees that the must-aliased heap locations are not reachable via may-aliases. In short, this is achieved by annotating the pointers in the structural environment as either may-alias pointers or must-alias pointers and making sure that must and may aliases never concretize to the same concrete pointer.

As before we introduce the syntax, the semantics in form of a concretization function and decorating structural well-formedness; for brevity we only present the changes to what has previously been presented. First, the syntax for structural values is extended to contain both structural may-alias pointers $\widehat{p}^{+}$ and structural must-alias pointers $\widehat{p}^{-}$.

$$\widehat{v} ::= \widehat{p}^{-} \mid \widehat{p}^{+} \mid \bullet$$

The separation between must and may aliases is achieved by demanding shared pointer valuations with pairwise disjoint codomains with the additional demand that the pointer valuations map all must-alias pointers to singleton sets, and to limit the concretization of $\bullet$ to non-pointer values. The demand that the pointer valuations are pairwise disjoint also for must-aliased pointers is not a restriction since two must-aliased pointers that concretize to the same concrete pointer is by definition may-aliased and may-aliases have priority over must-aliases in the merged alias information.

Definition 7.1 (May- and Must-alias sound pointer valuations) *Let $\widehat{p}$ range over structural may-alias pointers and structural must-alias pointers.*

$$\widehat{p} ::= \widehat{p}^{+} \mid \widehat{p}^{-}$$

A pointer valuation ξ is may- and must-alias sound if it has pairwise disjoint codomain that does not contain the null-pointer and maps all structural must-alias pointers to singleton sets.

$$\forall \widehat{p}_1, \widehat{p}_2 \in dom(\xi) \;.\; \widehat{p}_1 \neq \widehat{p}_2 \Longrightarrow \xi(\widehat{p}_1) \cap \xi(\widehat{p}_2) = \emptyset \wedge \forall \widehat{p^{-}} \in dom(\xi) \;.\; |\xi(\widehat{p^{-}})| = 1$$

Let ξ^* range over may- and must-alias sound pointer valuations.

The meaning of the combined may- and must-alias information is formulated in terms of a concretization function γ^*.

$$\gamma^*_{\xi^*}(\bullet) = \mathcal{V}_{\setminus p} \quad \gamma^*_{\xi^*}(\widehat{p}^{+}) = \xi^*(\widehat{p}^{+}) \cup \{nil\} \quad \gamma^*_{\xi^*}(\widehat{p}^{-}) = \xi^*(\widehat{p}^{-})$$

Similarly to before we form a decorated version of the structural values, stores and heaps by annotating $\bullet$ with a type. The concretization is changed accordingly to $\gamma^*_{\xi^*}(\bullet_\tau) = [\![\tau]\!]$, where $[\![int]\!]$ is the set of integers, $[\![nat]\!]$ the set of natural

numbers, and $[\![bool]\!]$ the set of booleans. The set of concrete environments associated with one particular combined structural environment is the union over all may- and must-alias sound pointer valuations.

$$\gamma^*(\widehat{E}) = \bigcup_{\xi^*} \{\gamma^*_{\xi^*}(\widehat{E})\}$$

The decorating structural well-formedness for the combined may- and must-alias information is immediate both for the undecorated syntax and the decorated syntax, using the rule for structural may pointers of Figure 5 for structural may pointers, and the rules for the undecorated $\bullet$, and the decorated $\bullet_\tau$ from Section 6 above.

$$\Omega \vdash \bullet : \tau \curvearrowright \bullet_\tau,\ \tau \in \{nat, int, bool\}$$

$$\frac{\tau_1 <: \tau_2}{\Omega \vdash \bullet_{\tau_1} : \tau_2 \curvearrowright \bullet_{\tau_2}}\ \tau_1, \tau_2 \in \{nat, int, bool\}$$

$$\frac{\Omega(\widehat{p}^-) = A_1 \quad A_1 <: A_2}{\Omega \vdash \widehat{p}^- : A_2 \curvearrowright \widehat{p}^-} \qquad \frac{\Omega(\widehat{p}^+) <: A \quad A <: \Omega(\widehat{p}^+)}{\Omega \vdash \widehat{p}^+ : A \curvearrowright \widehat{p}^+}$$

The structural well-formedness of the extended structural language have the same properties as the structural well-formedness for may-aliases of the previous section.

Lemma 7.1 (Stability of Type Decoration)

$$\Omega_{max} \vdash \widehat{E} : \Sigma \curvearrowright \widehat{E}_d \wedge \delta \vdash E : \Sigma \wedge E \in \gamma^*(\widehat{E}) \Longrightarrow E \in \gamma^*(\widehat{E}_d)$$

Proof 7.1 *First, $E \in \gamma^*(\widehat{E})$ implies the existence of ξ^* such that $E \in \gamma^*_{\xi^*}(\widehat{E})$. Now, Lemma B.2 gives us that $E \in \gamma^*_{\xi^*_{E,\Sigma}}(\widehat{E})$. The result is immediate from Lemma B.7, and Lemma B.9.*

Lemma 7.2 (Preservation of Well-formedness under Concretization)

$$\Omega \vdash \widehat{E} : \Sigma \wedge E \in \gamma^*_{\xi^*}(\widehat{E}) \Longrightarrow \delta_{\Omega,\xi^*} \vdash E : \Sigma$$

Proof 7.2 *Given that $E = (s, h)$ we must show that $\delta_{\Omega,\xi^*} \vdash s : \Sigma$, and that $\delta_{\Omega,\xi^*} \vdash h$. The result is immediate from Lemma C.3, and Lemma C.5 below.*

Merging May and Must Alias Information A structural may-alias environment, and a structural must-alias environment are *mergeable* with respect to a merge function f if the concretization of the merged result is conservative. Let $\widehat{E}^+$ range over structural may-alias environments, and $\widehat{E}^-$ range over structural must-alias environments.

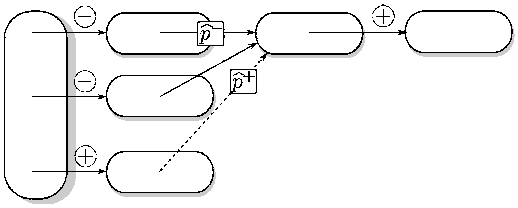
$$\gamma^+(\widehat{E}^+) \cap \gamma^-(\widehat{E}^-) \subseteq \gamma^*(f(\widehat{E}^+, \widehat{E}^-))$$

An abstract environment $\mathbb{E}$ is *mergeable* with respect to a merge function f, a may plugin $\mathcal{R}^+$, a must plugin $\mathcal{R}^-$, and an environment type Σ, if the extracted may- and must-alias environments are mergeable. We define the function *merge* as $merge(\mathbb{E}, \mathcal{R}^+, \mathcal{R}^-, \Sigma) = f(\eta^+(\mathbb{E}, \mathcal{R}^+, \Sigma), \eta^-(\mathbb{E}, \mathcal{R}^-, \Sigma))$ given that $\mathbb{E}$ is mergeable with respect to f, $\mathcal{R}^+$, $\mathcal{R}^-$, and Σ, and undefined otherwise. It is always possible to find merge functions, e.g., simply using the may-alias information is sound, possibly with the additional optimization that unique may-aliases are replaced by must-aliases as discussed above. For generality, in the following we parameterize over the merge function.

Strong Updates Using the combined may- and must-alias information we can create a flow-sensitive field update rule from the flow-insensitive counterpart. First, consider the type rule for field updates from Section 4 above.

$$\frac{\Sigma(x_1) = A \qquad \Sigma(x_2) = \tau \qquad \tau <: \Delta(A).f}{\Sigma \vdash x_1.f := x_2 \Rightarrow \Sigma, \bot_\Sigma}$$

Using the ideas outlined above, we can extract the merged alias-information; if the merge succeeds we know that the result is an accurate representation of the concrete environments reaching the command. Further, we know by construction that must-alias pointers form semi-isolated subgraphs in the heap in the sense that no may alias pointer points into the subgraphs, but may very well point out from it, as illustrated below, where + indicates may-aliases and − must-aliases. For the may-alias pointer $\widehat{p}^+$, represented by the dashed line in the figure, to point to the same position as the must-alias pointer $\widehat{p}^-$, they must be equal, i.e., $\widehat{p}^+ = \widehat{p}^-$, which is clearly not possible.



The basic idea is to perform the update in the type decorated structural representation of the environment and making sure that the new structural environment is well-formed with respect to the exit type. Let $upd_f(\widehat{p}^-, \widehat{v}, \widehat{E}) = (\widehat{s}_1, \widehat{h}_2)$ given that $\widehat{E} = (\widehat{s}_1, \widehat{h}_1)$, $\widehat{r}_1 = \widehat{h}_1(\widehat{p}^-)$, $\widehat{r}_2 = \widehat{r}_1[f \mapsto \widehat{v}]$, and $\widehat{h}_2 = \widehat{h}_1[\widehat{p}^- \mapsto \widehat{r}_2]$, i.e., the result of updating the field f with $\widehat{v}$ in the record pointed to by $\widehat{p}^-$ in $\widehat{E}$, defined identically for concrete environments. The rule for strong updates is defined as follows.

$$\frac{\begin{array}{c}\Omega_1 \vdash merge(\mathbb{M}_I(l), \mathcal{R}^+, \mathcal{R}^-, \Sigma_1) : \Sigma_1 \curvearrowright (\widehat{s}_1, \widehat{h}_1) \\ \widehat{s}_1(x_1) = \widehat{p}^- \qquad\qquad \widehat{s}_1(x_2) = \widehat{v} \\ \Omega_2 \vdash upd_f(\widehat{p}^-, \widehat{v}, (\widehat{s}_1, \widehat{h}_1)) : \Sigma_2\end{array}}{\Sigma_1 \vdash^{\mathbb{M}_I, \mathbb{M}_O, \mathcal{R}^+, \mathcal{R}^-} (x_1.f := x_2)^l \Rightarrow \Sigma_2, \bot_{\Sigma_1}}$$

The soundness of this rule relies on one important property for updates over must-aliases that expresses the soundness of the structural update.

Lemma 7.3 (Stability of Must-update under Concretization)

$$\{upd_f(p, v, E) \mid E \in \gamma^*_{\xi^*}(\widehat{E}), p \in \gamma^*_{\xi^*}(\widehat{p}^-), v \in \gamma^*_{\xi^*}(\widehat{v})\} = \gamma^*_{\xi^*}(upd_f(\widehat{p}^-, \widehat{v}, \widehat{E}))$$

Proof 7.3 *Since the store is unaffected by the update is suffices to show that the property holds for heaps, which is shown in the following lemma.*

Lemma 7.4 (Stability of Must-update under Heap Concretization)

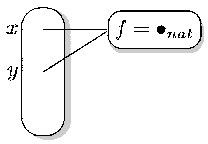
$$\{h[p \mapsto r] \mid h \in \gamma^*_{\xi^*}(\widehat{h}), p \in \gamma^*_{\xi^*}(\widehat{p}^-), r \in \gamma^*_{\xi^*}(\widehat{r})\} = \gamma^*_{\xi^*}(\widehat{h}[\widehat{p}^- \mapsto \widehat{r}])$$

Proof 7.4 *The proof relies on the facts that ξ^* has a pairwise disjoint codomain and $\gamma^*_{\xi^*}(\widehat{p}^-)$ is a singleton set not containing the null-pointer, since $\widehat{p}^-$ is a must-alias. Let $\{p\}$ be this set. Intuitively, the reasoning is as follows. On the left hand side we have that in all heaps in the concretization of $\widehat{h}$, p points to one of the concretizations of $\widehat{h}(\widehat{p}^-)$. Similarly, on the right hand side we have that in all heaps in the concretization of $\widehat{h}[\widehat{p}^- \mapsto \widehat{r}]$, p points to one of the concretizations of $\widehat{r}$. Now, if we take the concretization of $\widehat{h}$ and update p to point to a record in the concretization of $\widehat{r}$ then we get the same set as the concretization of $\widehat{h}[\widehat{p} \mapsto \widehat{r}]$.*

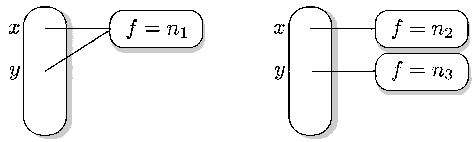
If $\widehat{p}^-$ was concretized to more than one concrete pointer, we would on the left hand side add heaps where only one of the concrete pointers is updated to point to records in the concretization of $\widehat{r}$, the other would still point to records in the concretization of $\widehat{h}(\widehat{p}^-)$.

If ξ^* did not have a pairwise disjoint codomain, p might be in the concretization of more structural pointers than $\widehat{p}$. This would require r to not only be in the concretization of $\widehat{r}$ but also in the concretization of each structural record pointed to by the additional structural pointers, something that in general would not be the case.

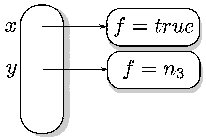
To see why the proof does not hold for may-aliases, it helps to illustrate why it holds for must-aliases. In fact, we can easily justify that $\{upd_f(p, v, E) \mid E \in \gamma^*_{\xi^*}(\widehat{E}), p \in \gamma^*_{\xi^*}(\widehat{p}^+), v \in \gamma^*_{\xi^*}(\widehat{v})\} \supseteq \gamma^*_{\xi^*}(upd_f(\widehat{p}^+, \widehat{v}, \widehat{E}))$, i.e., that the structural update is no longer guaranteed to be a sound approximation of the update. This comes from the fact that $\gamma^*_{\xi^*}(upd_f(\widehat{p}^+, \widehat{v}, \widehat{E}))$ only generates heaps where the records pointed to by all pointers $p \in \gamma^*_{\xi^*}(\widehat{p}^+)$ are updated whereas $\{upd_f(p, v, E) \mid E \in \gamma^*_{\xi^*}(\widehat{E}), p \in \gamma^*_{\xi^*}(\widehat{p}^+), v \in \gamma^*_{\xi^*}(\widehat{v})\}$ only updates the record pointer to by p. Consider the following example, where $\widehat{s} = \{x \mapsto \widehat{p}_1^+, y \mapsto \widehat{p}_1^+\}$, with $\widehat{h} = \{\widehat{p}_1^+ \mapsto \{f \mapsto \bullet_{nat}\}\}$.



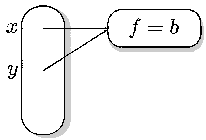
As described, the possible heaps concretized from $(\widehat{s}, \widehat{h})$ (ignoring null-pointers) can be illustrated as follows.



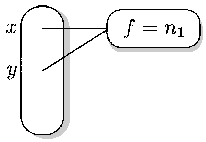
Updating the rightmost heap by $x.f := true$, i.e., $s_1 \mapsto \{x \mapsto p_1, y \mapsto p_2\}$, and $h_1 = \{p_1 \mapsto \{f \mapsto n_2\}, p \mapsto \{f \mapsto n_3\}\}$ results in a new heap $h_2 \mapsto \{p_1 \mapsto \{f \mapsto true\}, p \mapsto \{f \mapsto n_3\}\}$, illustrated below, i.e., where the record pointed to by y remains unchanged, since x and y were not aliased in this particular environment.



However, $upd_f(\widehat{p}^+, \widehat{v}, \widehat{E}) = (\widehat{s}, \widehat{h}_3)$, where $\widehat{h}_2 \mapsto \{\widehat{p}_1 \mapsto \{f \mapsto \bullet_{bool}\}$, will only produce environments of the following form.



This problem does not occur with must-aliases, since must-aliased locations are guaranteed to alias, i.e., $\widehat{s} = \{x \mapsto \widehat{p_1}, y \mapsto \widehat{p_1}\}$, with $\widehat{h} = \{\widehat{p_1} \mapsto \{f \mapsto \bullet_{nat}\}\}$ will only concretize to environments of the following structure, which are correctly modeled by the structural update.



With this, we prove the soundness of the strong update rule by proving its case in a preservation of types proof in the same way as in Section 6 above. Again, since the rule preserves width well-formedness under the assumption that the weakened command also preserves width well-formedness it can safely be added to the type system in Figure 3.

Lemma 7.5 (Preservation of Types of Strong Updates)

$$\Sigma_1 \vdash^{\mathbb{M}_1,\mathbb{M}_2,\eta^+,\eta^-} x_1.f := x_2 \Rightarrow \Sigma_2, \xi \wedge \\ is_c^{\mathcal{C}}(\mathbb{M}_I) \ \wedge \ os_c^{\mathcal{C}}(\mathbb{M}_O) \Longrightarrow \\ \forall E \in \mathcal{C}.\ \delta_1 \vdash E : \Sigma_1 \wedge \langle E, x_1.f := x_2 \rangle \rightarrow C \Longrightarrow \\ \exists \delta_2.\ \delta_2 \vdash C : \Sigma_2, \xi$$

Proof 7.5 *Assume an* $E \in \mathcal{C}$*, such that* (1) $\delta_1 \vdash E : \Sigma_1$*, and* (2) $\langle E, x_1.f := x_2\rangle \rightarrow C$*. We must show that* $\delta_2 \vdash C : \Sigma_2, \xi$ *for some* δ_2*.*

(2) *gives two possible cases: 1) the execution fails due to* x_1 *containing a null-pointer, and 2) the execution succeeds and* $C = upd_f(p, v, E) = (s, h[p \mapsto r[f \mapsto v]]$ *for* $E = (s, h)$*. The first case is a simple exception propagation, and we focus on the second case in the following.*

First, the soundness of the merge function gives us that together with (1) *and Lemma 7.1 gives that* $E \in \gamma^*_{\xi^*}(\widehat{s}_1, \widehat{h}_1)$ *for some pointer valuation* ξ^**, which also gives that* $p \in \gamma^*_{\xi^*}(\widehat{p}^-)$*, and* $v \in \gamma^*_{\xi^*}(\widehat{v})$*, since* $\widehat{p}^- = \widehat{s}(x_1)$*, and* $\widehat{v} = \widehat{s}(x_2)$*.*

We have that $upd_f(p, v, E) \in \gamma^*_{\xi^*}(upd_f(\widehat{p}^-, \widehat{v}, (\widehat{s}_1, \widehat{h}_1)))$ *from Lemma 7.3, and the result is immediate from Lemma 7.2.*

Example Use We end this section with a variation of the example of the previous section. Assuming $\Delta(A) = \{f : nat\}$ we saw how the following program was typable using structural weakening, by weakening the type of x and y to B, where $\Delta(B) = \{f : int\}$.

```
A x = new A; A y := x; x.f := 0; x.f := -1;
```

As discussed, structural weakening is limited to type changes that are supported by the subtype hierarchy. Thus, the following minor modification to the program makes the program untypable using structural weakening.

```
A x = new A; A y := x; x.f := 0; x.f := true;
```

As in the case above, even a simplistic alias analysis is able to determine that x and y are not only may-aliases but also must-aliases, and unaliased with all other locations. This means that the type rule for strong updates can be used to type $x.f := true$ which results in x and y getting the type B, where $\Delta(B) = \{f : bool\}$. The strong update is safe, since we know that x and y contain the same pointer in all program runs, and that this pointer is different from all other pointers.

8 Related Work

Using alias information to improve the precision of other analysis is widespread, e.g., [ABB06, CG93, CCL+96, LH98, PC04, FTA01, DF01, SWM99, WM01]. Common to most of these analyses is that they compute the needed alias information; our approach allows for the alias analysis to be parameterized, allowing different alias analyses to be plugged in with relative ease. Of the above mentioned work only the work on extending single static assignment (SSA) to non-scalar variables [CCL+96, CG93, LH98] uses parameterized information. It would be interesting to investigate to which extent the plugins framework could benefit the rest of the analyses.

Most closely related is the work by Smith, Walker and Morrisett [SWM99]. Therein they develop a pseudo-linear type system for alias types allowing for limited flow-sensitive types of aliased locations, and safe deallocation. With

respect to the type change, our work is a generalization of theirs; their flow-sensitivity is limited to linear types, and a very specific extension using dynamic type checking.

Also related is the work by Foster, Terauchi and Aiken [FTA01] on inferring flow-sensitive type qualifiers. Even though they limit their work to type qualifiers nothing seems to prohibit their method to be applied to the full types instead of only the qualifiers. Again our work generalizes their work with respect to the use of alias information for flow-sensitivity, since they restrict the flow-sensitivity to linear types; it would be interesting to see to which extent our ideas could be used to generalized their approach.

In [ABB06] Amtoft, Bandhakavi and Banerjee develop a hoare-style logic for reasoning about noninterference. In particular, the logic contains region assertions — a simple form of alias analysis — used to increase the precision of the analysis.

In [HS06] Hunt and Sands study a flow-sensitive type system for information flow security; their work shows us that there exists a most general lattice for each program — the powerset lattice of the variables — and that, for a simple imperative language with variables, one can form a type based transformation from the flow-sensitive type system to a flow-insensitive one. This suggests that flow-sensitivity might not be necessary for information flow security. The transformation does, however, rely on the ability of easily cloning the contents of variables, and statically allocating more variables to hold the values of different types. This is not always possible, or practical. For instance, in many JVM implementations the number of simultaneously live variable is limited.

With respect to the computation of alias information, see the work on shape analysis by Sagiv, Reps and Wilhelm [SRW96], or Walker and Morrisett [WM01] on recursive alias types. For a more recent result on shape analysis see the work by Yang et al. [YLB+08]. This work focuses on combining precision and scalability for use in the verification of device drivers and contains many interesting references to real world application of pointer analyses. For a more standard exposition of alias analysis see, e.g., [Deu94].

9 Conclusion

We have presented a way to allow for flow-sensitive types on the heap based on our plugin framework. In particular we have shown how may-alias information can be used to support *structural weakening*, where information about may-aliases is used to allow for a safe use of depth-subtyping in the subtyping rule, which made it possible to change the types of heap locations while retaining a uniform type view of all may-aliases, thus guaranteeing conformance with the concrete width well-formedness. Structural weakening only supports changing heap location type to more general types.

We have also shown how the combination of may- and must-alias information can be used to support strong updates, i.e., updates that do not have to follow the subtyping hierarchy. This was done by using a combined representation of

may-alias and must-alias information that guaranteed that no location was both must-and may-aliased with any other location.

In addition to this we have shown how may- and must-alias information can be extracted using our plugin framework. The use of the plugin framework has very much contributed to the generality of this work by forcing us to think abstractly about may- and must-aliases, and by allowing us a flexibility of exploring many different type rules with relative ease coming from the fact that the rules are free from the computation of alias information, only containing its usage.

Bibliography

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *POPL '06: Symposium on Principles of Programming Languages*, New York, NY, USA, 2006. ACM Press.

[CCL+96] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 253–267, London, UK, 1996. Springer-Verlag.

[CG93] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in ssa form. *SIGPLAN Not.*, 28(6):36–45, 1993.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, 29(6), 1994.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, New York, NY, USA, 2001. ACM.

[FTA01] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. Technical report, Berkeley, CA, USA, 2001.

[GH08] Tobias Gedell and Daniel Hedin. Abstract interpretation plugins for type systems. In *12th International Conference on Algebraic Methodology and Software Technology AMAST 2008*, LNCS, 2008.

[HS06] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06, Proceedings of the 33rd Annual. ACM SIGPLAN - SIGACT. Symposium. on Principles of Programming Languages*, January 2006.

[HS08] S. Hunt and D. Sands. Just forget it – the semantics and enforcement of information erasure. In *Programming Languages and Systems. 17th*

European Symposium on Programming, ESOP 2008, number 4960 in LNCS, 2008.

[LH98] Christopher Lapkowski and Laurie J. Hendren. Extended ssa numbering: Introducing ssa properties to language with multi-level pointers. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 128–143, London, UK, 1998. Springer-Verlag.

[PC04] Corneliu Popeea and Wei-Ngan Chin. A type system for resource protocol verification and its correctness proof. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 135–146, New York, NY, USA, 2004. ACM.

[Pie02] Benjamin C. Pierce, editor. *Types and Programming Languages*. MIT Press, 2002.

[Pie05] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–31, New York, NY, USA, 1996. ACM.

[SWM99] Frederick Smith, David Walker, and Greg Morrisett. Alias types. Technical report, Ithaca, NY, USA, 1999.

[WM01] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *TIC '00: Selected papers from the Third International Workshop on Types in Compilation*, pages 177–206, London, UK, 2001. Springer-Verlag.

[YLB+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.

A Cyclic Path Property

The path property $path(sl_1 \dots sl_2)$, expressing that the symbolic location sl_2 can be reached from the symbolic location sl_1 by a number of field references, is defined as follows where $fs(sl)$ returns the set of pointer fields of the symbolic location sl.

$$\begin{array}{lcl} path(sl) & \equiv & true \\ path(sl_1 \dots sl_n) & \equiv & path(sl_1 \dots sl_{n-1}) \wedge \\ & & \exists f \in fs(sl_{n-1}).\ sl_{n-1}.f = sl_n \end{array}$$

The cyclic property $cyclic(sl_1 \dots sl_n, \mathcal{R})$, expressing that there exists two unique locations sl_i and sl_j in $sl_1 \dots sl_n$ such that they are related by the plugin $\mathcal{R}$, is defined in the following way.

$$cyclic(sl_1 \dots sl_n, \mathcal{R}) \quad \equiv \quad \exists i, j \in [1 \dots n].\ i \neq j \ \wedge \ (sl_i, sl_j) \in \mathcal{R}$$

Finally, we define the cyclic path property $cyclicp(\mathcal{R})$, expressing that the plugin $\mathcal{R}$ has an upper limit n on the length of acyclic paths.

$$cyclicp(\mathcal{R}) \quad \equiv \quad \exists n.\ \forall sl_1 \dots sl_n.\ path(sl_1 \dots sl_n) \Longrightarrow cyclic(sl_1 \dots sl_n, \mathcal{R})$$

B Stability of Type Decoration

This section contains the proofs of stability of type decoration of the structural may-aliases of Section 6, and the combined structural may- and must-aliases of Section 7 — the former language is a sublanguage of the latter. In this section all pointer valuations are may- and must-alias sound, why the $*$ superscript is dropped from ξ^* throughout.

Definition B.1 (Ω order) *We define an order $\leq$ on abstract pointer typings as follows.*

$$\frac{\widehat{p} \in dom(\Omega_2).\ \Omega_1(\widehat{p}) <: \Omega_2(\widehat{p})}{\Omega_1 \leq \Omega_2}$$

The intuition behind the order is that bigger abstract pointer typings place less demands on structural environment.

Lemma B.1 (Maximal Ω) *For a given structural environment $\widehat{E}$, there exists a unique maximal (up to type constrained isomorphism) Ω_{max}, such that*

$$\Omega \vdash \widehat{E} : \Sigma \Longrightarrow \Omega \leq \Omega_{max} \wedge \Omega_{max} \vdash \widehat{E} : \Sigma$$

Proof B.1 *The intuition is that increasing abstract pointer typings place less demands on the structural environment, and that the maximal abstract-pointer typing for a given well-formed structural environment is given by Σ.*

First, it is clear that for a given Ω there exists a maximal abstract-pointer typing Ω_{max} obtainable by repeatedly choosing bigger pointer typings until no bigger exists in which $\widehat{E}$ is still Σ well-formed.

Now, assume that there exists two different maximal pointer typings Ω_{max_1} Ω_{max_2}*. It is clear that neither* $\Omega_{max_1} \leq \Omega_{max_2}$*, nor* $\Omega_{max_2} \leq \Omega_{max_1}$*, since in such case one of them would not be maximal. Thus, there exists at least one abstract pointer* $\widehat{p}$ *occurring at a symbolic location* l*, such that* $\Omega_{max_1}(\widehat{p}) <: \Sigma(l)$*, and* $\Omega_{max_2}(\widehat{p}) <: \Sigma(l)$*, but with* $\Omega_{max_1}(\widehat{p})$ *and* $\Omega_{max_2}(\widehat{p})$ *incomparable. Because of the use of width-subtyping we know that all shared fields of* $\Omega_{max_1}(\widehat{p})$*, and* $\Omega_{max_2}(\widehat{p})$ *must be equal.*

This gives us that $\Omega_{max_1}(\widehat{p})$ *has a field not in* $\Omega_{max_2}(\widehat{p})$ *and vise versa. However, the incompatible fields are not forced by* $\Sigma(l)$*, given by* $\Omega_{max_1}(\widehat{p}) <: \Sigma(l)$*, and* $\Omega_{max_2}(\widehat{p}) <: \Sigma(l)$*, which means* $\Omega_{max_1}(\widehat{p})$ *and* $\Omega_{max_2}(\widehat{p})$ *are not maximal — they can both be replaced by* $\Omega_{max_1}(\widehat{p}) \sqcap \Omega_{max_2}(\widehat{p}) <: \Sigma(l)$*.*

The maximal abstract pointer typing Ω_{max} *is easily obtained by a fixed-point iteration.*

Pointers not being live require special care when establishing stability of type decoration. To illustrate this, consider the case where $\widehat{s}(x) = \widehat{p}$, $\widehat{h}(\widehat{p}) = \{f \mapsto \bullet\}$, $\xi(\widehat{p}) = \{p_1, p_2\}$, $\Sigma(x) = A$ and $\Delta(A) = \{f : int\}$.

All heaps in the concretization of $\widehat{h}$ will have both p_1 and p_2 in its domain. In all heaps that are well-formed with respect to Σ, the well-formedness relation will require that the pointer that x contains will point to a record containing a value that is well-formed with respect to the type *int*. The other pointer will, however, not have any requirements placed on it since it will not be live.

However, in the concretization of the type decorated version of the heap, $\widehat{h}(\widehat{p}) = \{f \mapsto \bullet_{int}\}$, both p_1 and p_2 will be required to point to records containing values that are well-formed with respect to the type *int*.

This means that type decoration does not preserve concretizations for all pointer valuations. In order to work around this, we restrict ourselves to only consider the live pointers. This is reasonable to do, since a pointer that is not live is semantically safe to ignore. More, specifically, noting that the well-formedness relation only places requirements on pointers that are typed by Σ, we limit ourselves to only consider pointers that are live and that are given a type by Σ. We do this by defining Σ*-reachability*.

Definition B.2 (Σ-reachability) *We say that* p *is* Σ*-reachable in* E*, written* $p \in E_\Sigma$*, if there exists a* δ *such that* $\delta \vdash E : \Sigma$ *and there exists a symbolic location* l *such that* $\Sigma(l)$ *is defined and* $E(l) = p$*.*

When establishing stability of type decoration this is done with respect to a pointer valuation ξ whose codomain only consists of Σ-reachable pointers. We call such a ξ *minimal* and define it in the following way.

Definition B.3 (Minimal ξ) *For each environment* $E \in \gamma_\xi(\widehat{E})$ *such that* $\delta \vdash E : \Sigma$*, we define the minimal pointer valuation* $\xi_{E,\Sigma}$ *to be the sub-valuation of* ξ *that only contains the pointers* Σ*-reachable in* E*.*

$$\forall \widehat{p} \in dom(\xi), p \in \xi_{E,\Sigma}(\widehat{p}).\ p \in \xi(\widehat{p}) \wedge p \in E_\Sigma$$

Lemma B.2 ($\xi_{E,\Sigma}$ preserves E)

$$\Omega \vdash \widehat{E} : \Sigma \wedge E \in \gamma_\xi(\widehat{E}) \wedge \delta \vdash E : \Sigma \Longrightarrow E \in \gamma_{\xi_{E,\Sigma}}(\widehat{E})$$

Proof B.2 *By construction - $\xi_{E,\Sigma}$ contains all pointers that are live, and typed in E, but no else. Thus, it removes all demands on things that are not live or typed, while allowing for the same concretization of all live and typed locations.*

Lemma B.3 (Σ-reachable locations are well-formed)

$$\delta \vdash E : \Sigma \wedge E(l) = v \wedge \Sigma(l) = \tau \Longrightarrow \delta \vdash v : \tau$$

Proof B.3 *By straightforward induction on the symbolic location.*

Lemma B.4 (Σ-reachable pointers are well-formed)

$$\delta \vdash E : \Sigma \wedge p \in \gamma_{\xi_{E,\Sigma}} \Longrightarrow \exists A \ . \ \delta(p) = A$$

Proof B.4 *The result is immediate, since $\xi_{E,\Sigma}$ forms a subset of the Σ-reachable pointers.*

By definition if $p \in \gamma_{\xi_{E,\Sigma}}$ there exists a symbolic location l such that $E(l) = p$ and $\Sigma(l) = \tau$. From Lemma B.3 we have that $\delta \vdash p : \tau$, which gives us $\tau = A_1$, and $\delta(p) = A_2 <: A_1$.

Lemma B.5

$$\Omega_{max} \vdash \widehat{E} : \Sigma \wedge \delta \vdash E : \Sigma \wedge E \in \gamma_{\xi_{E,\Sigma}}(\widehat{E}) \Longrightarrow \\ p \in \xi_{E,\Sigma}(\widehat{p}) \wedge \widehat{p} \in dom(\Omega_{max}) \wedge \Longrightarrow \delta(p) <: \Omega_{max}(\widehat{p})$$

Proof B.5 *First we state some properties needed.*

1. *$\Omega_{max} \vdash \widehat{E} : \Sigma$ gives us that all symbolic locations l such that $\Sigma(l)$ is defined we either that $\widehat{E}(l) = \widehat{p}^+$, $\widehat{E}(l) = \widehat{p}^-$, or that $\widehat{E}(l) = \bullet$.*
2. *Let $L_{\widehat{p}^+}$ be the set of symbolic locations such that for $l \in L_{\widehat{p}^+}$, $\Sigma(l)$ is defined and $\widehat{E}(l) = \widehat{p}$. Since $E \in \gamma_\xi(\widehat{E})$ we have for each $p \in \xi(\widehat{p}^+)$ that the set $L_{\widehat{p}}$ such that $l \in L_p$, $\Sigma(l)$ is defined and $E(l) = p$ is a subset of $L_{\widehat{p}^+}$.*
3. *Let $L_{\widehat{p}^-}$ be the set of symbolic locations such that for $l \in L_{\widehat{p}^-}$, $\Sigma(l)$ is defined and $\widehat{E}(l) = \widehat{p}$. Since $E \in \gamma_\xi(\widehat{E})$ we have for the unique $p \in \xi(\widehat{p}^-)$ that the set L_p such that $l \in L_p$, $\Sigma(l)$ is defined and $E(l) = p$ is a equal to the set $L_{\widehat{p}^-}$.*
4. *From, $\Omega_{max} \vdash \widehat{E} : \Sigma$ we have that for all $l \in L_{\widehat{p}^+}$ it holds that $\Omega_{max}(\widehat{p}) <: \Sigma(l)$ and $\Sigma(l) <: \Omega_{max}(\widehat{p})$, i.e., all may-aliased locations have exactly the same type view (up to renaming). Together with $E \in \gamma_\xi(\widehat{E})$ this gives us that all occurrences of concrete pointers $p \in \xi(\widehat{p})$ have the same type view.*

5. *From,* $\Omega_{max} \vdash \widehat{E} : \Sigma$ *we have that for all* $l \in L_{\widehat{p}^-}$ *it holds that* $\Omega_{max}(\widehat{p}) <: \Sigma(l)$*, i.e., all must-aliased locations have compatible type views Together with* $E \in \gamma_\xi(\widehat{E})$ *this gives us that all occurrences of concrete pointers* $p \in \xi(\widehat{p})$ *have compatible type views.*

First, using $\delta \vdash E : \Sigma$ *and* $p \in \xi_{E,\Sigma}(\widehat{p})$*, Lemma B.4 gives us* $\delta(p) = A$ *for some A, i.e.,* $\delta(p)$ *is defined. The proof has two cases.*

May-alias case *Assume* $p \in \xi(\widehat{p}^+)$*, and* $\widehat{p}^+ \in dom(\Omega_{max})$*; from above we have that all* $l \in L_{\widehat{p}^+} \supseteq L_p$ *have exactly the same type view. Thus,* $\delta(p) <: \Sigma(l) = \Omega_{max}(\widehat{p}^+)$*.*

Must-alias case *Assume* $p \in \xi(\widehat{p}^-)$*, and* $\widehat{p}^- \in dom(\Omega_{max})$*; from above we have that* $l \in L_{\widehat{p}^+} = L_p$*. Furthermore, since* Ω_{max} *is maximal we know that* $\Omega_{max}(\widehat{p}^-) = \bigsqcap_{l \in L_{\widehat{p}^+}} \Sigma(l)$*. Thus, since for all* $l \in L_p$ *it holds* $\delta(p) <: \Sigma(l)$ *we have that* $\delta(p) <: \Omega_{max}(\widehat{p}^-) <: \Sigma(l)$*.*

Lemma B.6 (Stability of Value Type Decoration)

$$\Omega_{max} \vdash \widehat{v} : \tau_1 \curvearrowright \widehat{v}_d \wedge \delta \vdash v : \tau_2 \wedge \tau_2 <: \tau_1 \wedge v \in \gamma_\xi(\widehat{v}) \Longrightarrow v \in \gamma_\xi(\widehat{v}_d)$$

Proof B.6 *Assume* (1) $\Omega_{max} \vdash \widehat{v} : \tau_1 \curvearrowright \widehat{v}_d$*,* (2) $\delta \vdash v : \tau_2$*,* (3) $\tau_2 <: \tau_1$ *and* (4) $v \in \gamma_\xi(\widehat{v})$*.*

The proof continues by a case analysis of (1)*.*

case $\Omega_{max} \vdash \bullet : \tau_1 \curvearrowright \bullet_{\tau_1}$ *We must show that* $v \in \gamma_\xi(\widehat{v}_d) = [\![\tau_1]\!]$ *for the cases where* τ_1 *is one of int, nat or bool. This follows directly from* (2) *and* (3)*.*

case $\Omega_{max} \vdash \widehat{p} : \boldsymbol{A} \curvearrowright \widehat{p}$ *Since the decoration does not affect the concretization of pointers the result follows directly from* (4)*.*

Lemma B.7 (Stability of Store Type Decoration)

$$\Omega_{max} \vdash \widehat{s} : \Sigma \curvearrowright \widehat{s}_d \wedge \delta \vdash s : \Sigma \wedge s \in \gamma_\xi(\widehat{s}) \Longrightarrow s \in \gamma_\xi(\widehat{s}_d)$$

Proof B.7 *Assume* (1) $\Omega_{max} \vdash \widehat{s} : \Sigma \curvearrowright \widehat{s}_d$*,* (2) $\delta \vdash s : \Sigma$*, and* (3) $s \in \gamma_\xi(\widehat{s})$*.*

We must show that $\forall x \in dom(\widehat{s}_d)$*.* $s(x) \in \gamma_\xi(\widehat{s}_d(x))$*. We have that* (4) $\forall x \in dom(\widehat{s})$*.* $s(x) \in \gamma_\xi(\widehat{s}(x))$ *from* (3)*, and* (2) *gives us that* (5) $\forall x \in dom(\Sigma)$*.* $\delta \vdash s(x) : \Sigma(x)$*. Now,* (1) *gives us that* (6) $\forall (x, \tau) \in \Sigma$*.* $\Omega_{max} \vdash \widehat{s}(x) : \tau \curvearrowright \widehat{s}_d(x)$*,* (7) $dom(\widehat{s}) = dom(\widehat{s}_d)$*, and* (8) $\forall x \in dom(\widehat{s}) \setminus dom(\Sigma)$*.* $\widehat{s}_d(x) = \widehat{s}(x)$*.*

Thus, assuming $x \in dom(\widehat{s}_d)$*, we either have* $x \in dom(\widehat{s}) \setminus dom(\Sigma)$ *in which case we are done by* (8) *or* $x \in dom(\Sigma)$ *and* $x \in dom(\widehat{s})$ *by* (7)*. Now,* (6) *gives* $\Omega_{max} \vdash \widehat{s}(x) : \Sigma(x) \curvearrowright \widehat{s}_d(x)$*,* (5) *gives* $\delta \vdash s(x) : \Sigma(x)$*,* (4) *gives* $s(x) \in \gamma_\xi(\widehat{s}(x))$*, and we reach the conclusion via Lemma B.6.*

Lemma B.8 (Stability of Record Type Decoration)

$$\Omega_{max} \vdash \widehat{r} : \omega_1 \curvearrowright \widehat{r}_d \wedge \delta \vdash r : \omega_2 \wedge \omega_2 <: \omega_1 \wedge r \in \gamma_\xi(\widehat{r}) \Longrightarrow r \in \gamma_\xi(\widehat{r}_d)$$

Proof B.8 *Assume* (1) $\Omega_{max} \vdash \widehat{r} : \omega_1 \curvearrowright \widehat{r}_d$, (2) $\delta \vdash r : \omega_2$, (3) $\omega_2 <: \omega_1$, *and* (4) $r \in \gamma_\xi(\widehat{r})$.

To show $r \in \gamma_\xi(\widehat{r}_d)$ *we need to show that* $\forall f \in dom(\widehat{r}_d).\ r.f \in \gamma_\xi(\widehat{r}_d.f)$.

(1) *gives* (5) $\forall (f,\tau) \in \omega_1.\ \Omega_{max} \vdash \widehat{r}.f : \tau \curvearrowright \widehat{r}_d.f$, (6) $dom(\widehat{r}) = dom(\widehat{r}_d)$, *and* (7) $\forall f \in dom(\widehat{r}) \setminus dom(\omega).\ \widehat{r}_d(f) = \widehat{r}(f)$. (2) *gives* (8) $\forall (f,\tau) \in \omega_2.\ \delta \vdash r.f : \tau$. (3) *gives* (9) $\forall (f,\tau) \in \omega_1.\ \omega_2.f = \tau$.

Assume $f \in dom(\widehat{r}_d)$. (6) *gives that we have either* $f \in dom(\widehat{r}) \setminus dom(\omega_1)$ *and we are done by* (7)*, or* $f \in \widehat{r}$*, and* $f \in \omega_1$ *such that* $\Omega_{max} \vdash \widehat{r}.f. : \omega_1.f. \curvearrowright \widehat{r}_d.f$. *Now,* (9) *gives us that* $\omega_2.f = \omega_1.f$*, and* (8) *gives* $\delta \vdash r.f : \omega_2.f$. *With this Lemma B.6 allows us to conclude.*

Lemma B.9 *Stability of Heap Type Decoration*

$$\Omega_{max} \vdash (s, \widehat{h}) : \Sigma \curvearrowright (\widehat{s}_d, \widehat{h}_d) \wedge \delta \vdash (s,h) \wedge h \in \gamma_{\xi_{E,\Sigma}}(\widehat{h}) \Longrightarrow h \in \gamma_{\xi_{E,\Sigma}}(\widehat{h}_d)$$

Proof B.9 *Assume* (1) $\Omega_{max} \vdash \widehat{h} \curvearrowright \widehat{h}_d$, (2) $\delta \vdash h$, *and* (3) $h \in \gamma_{\xi_{E,\Sigma}}(\widehat{h})$. (1) *gives* (4) $\forall (\widehat{p}, A) \in \Omega_{max}.\ \Omega_{max} \vdash \widehat{h}(\widehat{p}) : \Delta(A) \curvearrowright \widehat{h}_d(\widehat{p})$, (5) $dom(\widehat{h}) = dom(\widehat{h}_d)$, *and* (6) $\forall \widehat{p} \in dom(\widehat{h}) \setminus dom(\Omega)_{max}.\ \widehat{h}_d(\widehat{p}) = \widehat{h}(\widehat{p})$. (2) *gives* $\forall (p, A) \in \delta.\ \delta \vdash h(p) : \Delta(A)$, *and* (3) *gives* $\forall \widehat{p} \in dom(\widehat{h}), p \in \xi_{E,\Sigma}(\widehat{p}).\ h(p) \in \gamma_{\xi_{E,\Sigma}}(\widehat{h}(\widehat{p}))$.

We must show that $\forall \widehat{p} \in dom(\widehat{h}_d), p \in \xi_{E,\Sigma}(\widehat{p}).\ h(p) \in \gamma_{\xi_{E,\Sigma}}(\widehat{h}_d(\widehat{p}))$. *Assume* $\widehat{p} \in dom(\widehat{h}_d)$ *and* $p \in \xi_{E,\Sigma}(\widehat{p})$. (5) *gives either* $\widehat{p} \in dom(\widehat{h}) \setminus dom(\Omega_{max})$ *and we are done by* (6) *or* $(\widehat{p}, A_1) \in \Omega_{max}$ *and thus that* $\Omega_{max} \vdash \widehat{h}(\widehat{p}) : \Delta(A_1) \curvearrowright \widehat{h}_d(\widehat{p})$ *by* (4). *From Lemma B.5 we have that* $\delta(p) <: A_1$*, which implies that* δ *is defined for* p*, i.e.,* $\delta(p) = A_2$ *for some* A_2. *Now,* (2) *gives us* $\delta \vdash h(p) : \Delta(A_2)$*, and we are done by Lemma B.8.*

Lemma B.10 (Stability of Type Decoration)

$$\Omega_{max} \vdash\!: \widehat{E} : \Sigma \curvearrowright \widehat{E}_d \wedge \delta \vdash\!: E : \Sigma \wedge E \in \gamma(\widehat{E}) \Longrightarrow E \in \gamma(\widehat{E}_d)$$

Proof B.10 *First,* $E \in \gamma(\widehat{E})$ *implies the existence of* ξ *such that* $E \in \gamma_\xi(\widehat{E})$. *Now, Lemma B.2 gives us that* $E \in \gamma_{\xi_{E,\Sigma}}(\widehat{E})$. *The result is immediate from Lemma B.7, and Lemma B.9.*

Lemma B.11 *Type Decoration Preserves Well-formedness*

$$\Omega \vdash \widehat{E}_1 : \Sigma \curvearrowright \widehat{E}_2 \Longrightarrow \Omega \vdash \widehat{E}_2 : \Sigma$$

Proof B.11 *The result is immediate from inspecting the rules and noting that the only decoration takes place in the well-formedness rule for* $\bullet$ *and that the decoration is the type demanded by well-formedness, i.e.,* $\Omega \vdash \bullet : int : \bullet_{int}$.

C Preservation of Well-formedness under Concretization

This section contains the proofs that well-formedness is preserved by concretization of the structural may-aliases of Section 6, and the combined structural may- and must-aliases of Section 7. In this section all pointer valuations are may- and must-alias sound, why the $*$ superscript is dropped from ξ^* throughout.

Lemma C.1 *Preservation of Well-formedness under Concretization*

$$\Omega \vdash \widehat{E} : \Sigma \wedge E \in \gamma_\xi(\widehat{E}) \Longrightarrow \delta_{\Omega,\xi} \vdash E : \Sigma$$

Proof C.1 *Given that $E = (s, h)$ we must show that $\delta_{\Omega,\xi} \vdash s : \Sigma$, and that $\delta_{\Omega,\xi} \vdash h$. The result is immediate from Lemma C.3, and Lemma C.5 below.*

Lemma C.2 *Preservation of Well-formedness under Value Concretization*

$$\Omega \vdash \widehat{v} : \tau \wedge v \in \gamma_\xi(\widehat{v}) \Longrightarrow \delta_{\Omega,\xi} \vdash v : \tau$$

Proof C.2 *We proceed by a case analysis on $\widehat{v}$.*

case $\widehat{v} = \bullet_{\tau_1}$ *We have that $v \in [\![\tau_1]\!]$ from $v \in \gamma_\xi(\widehat{v})$, and that $\tau_1 <: \tau$ from $\Omega \vdash \widehat{v} : \tau$, and the result is immediate.*

case $\widehat{v} = \widehat{p}^+$ *We have that $v = p \in \xi(\widehat{p}^+)$ from $v \in \gamma_\xi(\widehat{p}^+)$, and that $\Omega(\widehat{p}^+) = \tau = A$ for some A. By definition $\delta_{\Omega,\xi}(p) = \Omega(\widehat{p}^+)$ given $p \in \gamma_\xi(\widehat{p}^+)$, and the result is immediate.*

case $\widehat{v} = \widehat{p}^-$ *We have that $v = p \in \xi(\widehat{p}^-)$ from $v \in \gamma_\xi(\widehat{p}^-)$, and that $\Omega(\widehat{p}^-) <: \tau = A$ for some A. By definition $\delta_{\Omega,\xi}(p) = \Omega(\widehat{p}^-)$ given $p \in \gamma_\xi(\widehat{p}^-)$, and the result is immediate.*

Lemma C.3 (Preservation of Well-formedness under Store Concretization)

$$\Omega \vdash \widehat{s} : \Sigma \wedge s \in \gamma_\xi(\widehat{s}) \Longrightarrow \delta_{\Omega,\xi} \vdash ws : \Sigma$$

Proof C.3 *We must show that $(x, \tau) \in \Sigma$. $\delta_{\Omega,\xi} \vdash s(x) : \tau$. Assume $(x, \tau) \in \Sigma$. We have that $\Omega \vdash \widehat{s}(x) : \tau$ from $\Omega \vdash \widehat{s} : \Sigma$, and $s(x) \in \gamma_\xi(\widehat{s}(x))$ from $s \in \gamma_\xi(\widehat{s})$. The result is immediate from Lemma C.2.*

Lemma C.4 (Preservation of Well-formedness under Record Concretization)

$$\Omega \vdash \widehat{r} : \omega \wedge r \in \gamma_\xi(\widehat{r}) \Longrightarrow \delta_{\Omega,\xi} \vdash r : \omega$$

Proof C.4 *We must show that $(f, \tau) \in \omega$. $\delta_{\Omega,\xi} \vdash r.f : \tau$. Assume $(f, \tau) \in \omega$. We have that $\Omega \vdash \widehat{r}.f : \tau$ from $\Omega \vdash \widehat{r} : \omega$, and $r.f \in \gamma_\xi(\widehat{r}.f)$ from $r \in \gamma_\xi(\widehat{r})$. The result is immediate from Lemma C.2.*

Lemma C.5 (Preservation of Well-formedness under Heap Concretization)

$$\Omega \vdash \widehat{h} \wedge h \in \gamma_\xi(\widehat{h}) \Longrightarrow \delta_{\Omega,\xi} \vdash h$$

Proof C.5 *We must show that* $(p, A) \in \delta_{\Omega,\xi}$. $\delta_{\Omega,\xi} \vdash h(p) : \Delta(A)$. *Assume* $(p, A) \in \delta_{\Omega,\xi}$. *By definition of* $\delta_{\Omega,\xi}$ *we have that there exists an abstract structural pointer (must or may)* $\widehat{p}$ *such that* $p \in \xi(\widehat{p})$, *and* $\Omega(\widehat{p}) = A$. *We have that* $\Omega \vdash \widehat{h}(\widehat{p}) : \Delta(A)$ *from* $\Omega \vdash \widehat{h}$, *and from the fact that* ξ *has pairwise disjoint codomain we have that* $\widehat{p}$ *is unique, which together* $h \in \gamma_\xi(\widehat{h})$, *gives us* $h(p) \in \gamma_\xi(\widehat{h}(\widehat{p}))$ *and the result is immediate from Lemma C.4.*