Using SMT Solvers for False-Positive Elimination in Static Program Analysis

Maximilian Junker

Master’s Thesis

University of Augsburg
Ludwig Maximilian University of Munich
Technical University of Munich

Supervisors
Prof. Dr. Alexander Knapp
Prof. Dr. Bernhard Bauer

External Supervisors
Dr. Ralf Huuck
Dr. Ansgar Fehnker
Using SMT Solvers for False-Positive Elimination in Static Program Analysis

Maximilian Junker

November 22, 2010
Abstract

The aim of static program analysis is to find program defects quickly, reliably, and exhaustively. One possibility to reach this goal is to perform the analysis on abstractions of the inspected programs. The problem with this approach is that it may produce so-called false-positives, i.e. error warnings that are not true bugs. In this thesis, we present a method to eliminate many false-positives that arise in model checking-based static analysis, as it is pursued in the Goanna tool. We develop a language for describing paths through programs written in C and use weakest preconditions to transform those paths into first-order formulas for which we can decide satisfiability using an SMT solver. Based on this, we show how to iteratively eliminate false-positives using automata-based representations of programs. We implemented a prototype of the procedure in the framework of Goanna and report on the results of first experiments.
Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 22. November 2010
Maximilian Junker
1 Introduction

2 Preliminaries
   2.1 Static Program Analysis
       2.1.1 A Short Overview
       2.1.2 Model Checking-Based Static Analysis
       2.1.3 The Static Analysis Tool Goanna
   2.2 First-Order Logic
   2.3 SAT Modulo Theories
       2.3.1 What is SMT?
       2.3.2 The Theory behind SMT-Solving
       2.3.3 Summary: What SMT Solvers Can Do and Why

3 False-Positive Elimination in Goanna
   3.1 Interval based False-Positive Elimination
   3.2 Problems With the Current Approach
   3.3 SMT based False-Positive Elimination

4 A Language for Specifying Program Paths
   4.1 Syntax
   4.2 Semantics
       4.2.1 Transformation
       4.2.2 Memory Model
       4.2.3 Type System
       4.2.4 Semantics of Expressions
       4.2.5 Semantics of Statements
       4.2.6 Properties of wp

5 Path Reduction
   5.1 Finite and Infinite Automata
   5.2 Representing Programs as Automata
   5.3 Initial Abstraction and Refinement Loop
   5.4 Building Observers From Infeasible Cores
   5.5 Computing Infeasible Cores
Chapter 1

Introduction

It is a well-known fact that defects in software systems may have catastrophic consequences. There are countless examples where small and arguably avoidable programming errors have led to situations where large sums of money or even the health of people were put at risk. Therefore, the research about methods that help constructing safe software has intensified over the last decades. Those methods include purely organisational measures to insure quality but also procedures that work directly on the software, like (automatic) testing, verification and particularly static program analysis. Especially in the last years there has been considerable success in applying these methods on real-world problems. Renowned software companies now routinely use static analysis techniques on their code-base [4]. Through advances in algorithms as well as growing computing power, static analysis techniques are now able to find complex defects in the source code with minimal effort from the side of the developers. However, static analysis techniques are often imprecise. In general, neither do they guarantee that there are no bugs besides the ones they claim to have found, nor can one be sure that the found bugs are real. That means the task to assess the issued warnings falls back onto the developer. But with the increasing complexity of the bugs that those techniques can uncover, this assessment is getting more and more difficult. In large software projects, developers may be forced to spend a lot of time assessing a warning of a static analysis tool just to find out that the claimed bug is not real.

Therefore, it is vital for static analysis not only to find many complex bugs, but also to assure that the majority of those are not false-positives. There are different approaches for the minimization of those cases. Besides a careful design of the analysis method, one may use a filter after or inside the analysis. Such a false-positive elimination procedure analyzes the bug reports and sorts out those that are highly likely to be false-positives.

In this thesis we develop such a false-positive elimination procedure for the static analysis tool Goanna [25].
In Chapter 2, we describe the basic concepts that we use in the rest of the thesis. Especially we fix the notation for the central mathematical vehicle that we use later: first-order logic.

Goanna uses a false-positive elimination procedure which we will describe in Chapter 3. We will show what the problems are and give an overview of our revised false-positive elimination procedure.

In Chapter 4, we introduce a language that allows us to specify a path through a program that is written in the programming language C. We give the syntax of this language and after that develop an axiomatic semantics for it. This involves the definition of a memory model to enable reasoning about language features like pointers. The semantics will be based on the well-known concept of weakest preconditions.

The heart of our false-positive elimination procedure will be outlined in Chapter 5. It is an automata-based approach that is similar to the one currently implemented in Goanna but is built upon the semantics of the path language. We will give proofs of the correctness of the procedure relative to the semantics of the path language.

In Chapter 6, we provide some details about the implementation and first results of running the procedure on the source code of the open source projects Wireshark and Dovecot.

The remaining chapters show what works are related to ours and summarize the whole thesis.
Chapter 2

Preliminaries

2.1 Static Program Analysis

2.1.1 A Short Overview

This thesis is a contribution to the broad field of static program analysis. This term is used to describe automatic analysis techniques that operate on representations of programs (often source code or some intermediate language) without actually executing the program. “Automatic” in this context means that static analysis tools normally do not require any interaction with a person. Static analysis is commonly used for compiler optimizations (e.g. to avoid unnecessary computations and thus create more efficient code) and to find bugs in programs (e.g. a possible null-pointer dereferencing or potential deadlocks). However, static analysis can also be used to discover more complex security threats (e.g. possible SQL-injections). Static analysis techniques are typically based on some mathematical model of the program. Depending on the usage scenario there are different approaches using different mathematical frameworks. Popular techniques include abstract interpretation, data-flow analysis or model checking. They may even be combined or information obtained by using one approach may be used to improve a procedure based on another approach. An common problem in all those approaches is that a program in general may exhibit an infinite number of states and behaviours. Therefore static analysis is often based on some finite abstraction of a program. Again depending on the problem, such an abstraction may be an under-approximation (i.e. the abstract program has less states and/or behaviours than the original) or an over-approximation (i.e. the abstract program has more states and/or behaviours than the original). In the case of Goanna – which is the analysis tool we will describe in the next chapter and which provides the context of this thesis – we have the case of an over-approximation. A major issue in this case is that it is possible that a reported bug cannot appear in the original program. Those bugs are commonly called false-positives and should be avoided.
2.1 Static Program Analysis

2.1.2 Model Checking-Based Static Analysis

We now describe the model checking-based approach for static analysis in greater detail as this is the one pursued in Goanna.

The model checking problem is the problem to decide if some system description, typically given as a Kripke structure, fulfills (is a model of) a specification, given, for example, as a formula in temporal logic. In the case where a system description is not a model of the specification one would like to have a proof that this is in fact true. Such a proof is often given as a counter-example, i.e. a run of the described system that violates the specification. The system in our case is some program and a counter-example a trace through the program. We may now perform static analysis by formulating the properties of interest as formulas in temporal logic. We assume we have a system description that is an over-approximation of the original program and therefore, as described above, there is the problem of spurious warnings.

A popular approach to tackle this problem is a procedure called Counter-Example-Guided Abstraction Refinement (CEGAR) [17]. The idea here is to start with a very coarse abstraction of the program that may be checked quickly but may exhibit spurious behaviour. If the model checker does not find a violation of the specification, the specification is also not violated in the original program. If however the specification is indeed violated, one needs to check if the violation is only an artifact due to the approximation. This can be done using the counter-example that the model checker provides. If the violation is valid also in the original program, the procedure has found a real bug. Otherwise the current system description is too coarse and needs to be refined in some way such that the counter-example does not represent a behaviour of the refined system any more. This refinement also makes use of the counter-example, which explains the name of the procedure. The refined system is then again subjected to model checking. This loop is executed until a valid violation is found or no more violations are found at all. Depending on the details of the refinement the procedure is not guaranteed to terminate and the number of loops may have to be bounded.

2.1.3 The Static Analysis Tool Goanna

The Goanna Static Analysis Tool [25] adopts an approach that is very similar to the CEGAR approach described above. Goanna analyzes the source code of a program written in the C programming language. In order to perform software model checking, a mathematical system description is needed. Goanna pursues a syntax-based approach. It uses an annotated Control Flow Graph (CFG) of a program as model of the program. The specification is then encoded as a CTL formula over the CFG. We refer to [26] for a detailed description of how the CFG is created and how the properties
are encoded. Typical bugs that can be found with Goanna are uninitialized variables or null-pointer dereferencing. However, as the annotated CFG is a very coarse abstraction of the actual program (as it contains not much information about data flow) the problem of false-positives arises. Therefore Goanna implements Counterexample-Guided Path Reduction [27] that is quite similar to the CEGAR approach. Goanna uses an interval based semantics of the program to build a refined model based on the counterexample of the model checker. We describe the procedure in more detail together with its problems in Chapter 3, as the aim of this thesis is exactly to propose an alternative approach for false-positive elimination.

2.2 First-Order Logic

As we take recourse to first-order logic in many parts of this thesis, we introduce the basic concepts and notations. For our presentation we use a many-sorted framework. A comprehensive description of first-order logic can be found in [24]. Most of the formalization of substitutions are adopted from [5].

\textbf{Definition 2.1.} A signature \( \Sigma = (S, F, P) \) consists of

- a finite set \( S \) of sorts,
- a finite set \( F \) of function symbols \( f : s_1, \ldots, s_n \to s \) where \( s_1, \ldots, s_n \) are the argument sorts and \( s \) is the result sort,
- a finite set \( P \) of predicate symbols \( p : s_1, \ldots, s_n \).

\textbf{Definition 2.2.} Let \( \Sigma = (S, F, P) \) be a signature and \( X = \bigcup_{s \in S} X_s \) a set of variables for \( \Sigma \). Then the sets \( \mathcal{T}(\Sigma, X)_s \) of terms of sort \( s \) over \( \Sigma \) and \( X \) are given by

- \( x \in X_s \) is a term of sort \( s \)
- if \( t_1, \ldots, t_n \) are terms of sorts \( s_1, \ldots, s_n \) and \( f : s_1, \ldots, s_n \to s \in F \), then \( f(t_1, \ldots, t_n) \) is a term of sort \( s \)

With \( \mathcal{T}(\Sigma, X) \) we denote all terms over \( \Sigma \) and \( X \) of any sort.

\textbf{Definition 2.3.} For a signature \( \Sigma = (F, S, P) \) and a set of variables \( X \) the set of first order formulas \( \mathcal{F}(\Sigma, X) \) is given by

- \( \text{true, false} \in \mathcal{F}(\Sigma, X) \)
- \( t_1 = t_2 \in \mathcal{F}(\Sigma, X) \) if \( t_1, t_2 \in \mathcal{T}(\Sigma, X)_s \) for some \( s \in S \)
- \( p(t_1, \ldots, t_n) \in \mathcal{F}(\Sigma, X) \) if \( p : s_1, \ldots, s_n \in P \) and \( t_i \in \mathcal{T}(\Sigma, X)_{s_i} \)
2.2 First-Order Logic

- if \( \varphi, \psi \in \mathcal{F}(\Sigma, X) \), then also \((\varphi \lor \psi), (\neg \varphi) \in \mathcal{F}(\Sigma, X)\)
- if \( \varphi \in \mathcal{F}(\Sigma, X) \) and \( x \in X \) then also \((\exists x. \varphi) \in \mathcal{F}(\Sigma, X)\)

Furthermore we use the following abbreviations:

\[
\varphi_1 \land \varphi_2 := \neg(\neg \varphi_1 \lor \neg \varphi_2)
\]
\[
\varphi_1 \Rightarrow \varphi_2 := \neg \varphi_1 \lor \varphi_2
\]
\[
\varphi_1 \iff \varphi_2 := (\varphi_1 \Rightarrow \varphi_2) \land (\varphi_2 \Rightarrow \varphi_1)
\]
\[
\forall x. \varphi := \neg \exists x. \neg \varphi
\]

Remark on the use of the equality symbol: We have two different types of equality signs. One belongs to our meta language. We will denote this type of equality with the normal equality symbol “=”. On the other hand, we have equality in the logic. We will use the “\(\Leftrightarrow\)” symbol for this type of equality.

**Definition 2.4.** Let \( \Sigma \) be a signature and \( X \) be a set of variables. A \((\Sigma, X)\)-substitution \( \sigma : \bigcup_{s \in S} (X_s \rightarrow \mathcal{I}(\Sigma, X)_s) \) is a function such that \( \sigma(x) \neq x \) for only finitely many \( x \). The domain \( \text{dom}(\sigma) \) of a substitution \( \sigma \) is the set \( \{ x \in X \mid \sigma(x) \neq x \} \). The range \( \text{rng}(\sigma) \) of a substitution \( \sigma \) is the set \( \{ \sigma(x) \mid x \in \text{dom}(\sigma) \} \). If \( \text{dom}(\sigma) = \{ x_1, \ldots, x_n \} \), we may write \( \sigma \) as \( \{ x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n) \} \).

**Definition 2.5.** A term extension \( \hat{\sigma} : \mathcal{I}(\Sigma, X) \rightarrow \mathcal{I}(\Sigma, X) \) of a substitution \( \sigma \) is a mapping such that

- \( \hat{\sigma}(x) = \sigma(x) \) if \( x \in X \)
- \( \hat{\sigma}(f(t_1, \ldots, t_n)) = f(\hat{\sigma}(t_1), \ldots, \hat{\sigma}(t_n)) \)

Similar, a formula extension \( \tilde{\sigma} : \mathcal{F}(\Sigma, X) \rightarrow \mathcal{F}(\Sigma, X) \) of \( \sigma \) is a mapping given by recursively applying \( \hat{\sigma} \) to terms and renaming bound variables if there is a naming conflict. We will identify \( \hat{\sigma} \) and \( \tilde{\sigma} \) with \( \sigma \) if it is clear from the context that an extension is meant.

**Definition 2.6.** The composition \( \sigma \tau \) of two substitutions \( \sigma \) and \( \tau \) is the substitution

\[
(\tau \sigma)(x) := \hat{\tau}(\sigma(x))
\]

**Remark:** It is common in the literature to write substitutions behind formulas to improve readability. We will adopt this notation at some occasions. In those cases \( \varphi \sigma_1 \ldots \sigma_n \) is equivalent to \( (\sigma_n \ldots \sigma_1)(\varphi) \).

**Definition 2.7.** The restriction \( \sigma|_Y \) of a substitution \( \sigma \) to a set of variables \( Y \subseteq X \) is a substitution such that
2.2 First-Order Logic

σ|_Y(x) = \begin{cases} σ(x) & \text{if } x \in Y \\ x & \text{if } x \notin Y \end{cases}

The application \((σ ∘ τ)\) of a substitution \(τ\) on a substitution \(σ\) is defined as \((σ ∘ τ) := (τσ)|_{dom(σ)}\)

**Definition 2.8.** The empty substitution \(σ_{∅}\) is the substitution such that \(dom(σ_{∅}) = ∅\).

**Definition 2.9.** An algebra \(A = ((A_s)_{s ∈ S}, (f^A)_{f ∈ F}, (p^A)_{p ∈ P})\) over a signature \(Σ = (S, F, P)\) consists of

- non-empty carrier sets \(A_s\) for each sort \(s ∈ S\),
- functions \(f^A : A_{s_1} × ⋯ × A_{s_n} → A_s\) for each of the signature's function symbols \(f : s_1, ⋯, s_n → s ∈ F\),
- predicates \(p^A \subseteq A_{s_1} × ⋯ × A_{s_n}\) for each of the signature's predicate symbols \(p : s_1, ⋯, s_n ∈ P\).

For a signature \(Σ\), \(Alg(Σ)\) denotes the set of \(Σ\)-algebras.

**Definition 2.10.** A valuation \(v : \bigcup_{s ∈ S} (X_s → A_s)\) is a mapping that assigns a value \(v(x) ∈ A_s\) to each variable symbol \(x ∈ X_s\).

**Definition 2.11.** A valuation update \(v[x ↦ a]\) of \(v\) for \(x ∈ X_s\) and \(a ∈ A_s\) is given by

\[ v[x ↦ a](y) = \begin{cases} a & \text{if } y = x \\ v(y) & \text{otherwise} \end{cases} \]

**Definition 2.12.** Let \(A\) be a \(Σ\)-algebra and let \(v\) be a valuation. Then, the semantics \(T[t]Av\) of a term \(t ∈ T(Σ, X)\) is the value in \(A_s\) such that

- \(T[x]Av = v(x), \text{ if } x ∈ X\)
- \(T[f(t_1, ⋯, t_n)]Av = f^A(T[t_1]Av, ⋯, T[t_n]Av)\) for \(f : s_1, ⋯, s_n → s ∈ F\) and \(t_i ∈ T(Σ, X)_{s_i}\)

**Definition 2.13.** For an algebra \(A\) and a valuation \(v\) the semantics of a formula \(Formula\varphi\) is the truth value \(b ∈ \{tt, ff\}\) such that

- \(Formula[true]Av = tt, Formula[false]Av = ff\)
- \(Formula[t_1 = t_2]Av = tt \text{ iff } T[t_1]Av = T[t_2]Av\)
2.2 First-Order Logic

- $F[p(t_1, \ldots, t_n)]A v = tt$ iff $(T[t_1]A v, \ldots, T[t_n]A v) \in p^A$
- $F[\neg \varphi]A v = tt$ iff not $F[\varphi]A v = tt$
- $F[\varphi_1 \lor \varphi_2]A v = tt$ iff $F[\varphi_1]A v = tt$ or $F[\varphi_2]A v = tt$
- $F[\exists x. \varphi]A v = tt$ iff there is an $a \in A_x$ such that $F[\varphi]A v\{x \mapsto a\} = tt$, for $x \in X_s$

**Definition 2.14.** A $\Sigma$-formula $\varphi$ is valid in a $\Sigma$-algebra $A$ and a valuation $v$, denoted by $A, v \models \varphi$, if $F[\varphi]A v = tt$.

$\varphi$ is valid in $A$ if it is valid in $A$ for every valuation.

$\varphi$ is satisfiable in an algebra $A$ if there is a valuation $v$ such that $A, v \models \varphi$.

$\varphi$ is satisfiable if it is satisfiable in some $\Sigma$-algebra.

**Definition 2.15.** A substitution $\sigma^{-1}$ is a left-inverse of a substitution $\sigma$ if $(\sigma^{-1} \sigma)(x) = x$ is valid.

**Definition 2.16.** For a signature $\Sigma$, a $\Sigma$-theory $T \subseteq \text{Alg}(\Sigma)$ is a set of $\Sigma$-Algebras. A formula $\varphi$ is satisfiable in $T$ if it is satisfiable in an algebra $A \in T$.

We now establish a well-known fact about substitutions in formulas. As the proof is not complicated but rather lengthy we do not reproduce it here but instead refer to [24].

**Lemma 2.1** (Substitution Lemma). The formula $\varphi\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is true in the algebra $A$ and the valuation $v$ if it is true in $A$ and an updated valuation where $x_i$ has the value of $t_i$:

$$F[\varphi\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}]A v = F[\varphi]A v\{x_1 \mapsto T[t_1]A v, \ldots, x_n \mapsto T[t_n]A v\}$$

We do, however, prove the following lemmas which are important for the results of Chapter 5.

**Lemma 2.2.** If $\varphi\sigma$ is satisfiable for a formula $\varphi$ and a substitution $\sigma$, then also $\varphi$ is satisfiable.

**Proof.** Let $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. There is an algebra $A$ and a valuation $v$ such that $A, v \models \varphi\sigma$. By Lemma 2.1 we get that

$$F[\varphi\sigma]A v = F[\varphi]A v'$$

where $v' = \{x_1 \mapsto T[t_1]A v, \ldots, x_n \mapsto T[t_n]A v\}$.

Because of this equivalence we have $A, v' \models \varphi$. Therefore $\varphi$ is satisfiable.

□
Lemma 2.3. Let $\sigma$ and $\tau$ be substitutions. Then

$$\tau\sigma = (\sigma \triangleleft \tau)\tau$$

if $\text{dom}(\tau) \cap \text{dom}(\sigma) = \emptyset$ and $\text{var}(\text{rng}(\tau)) \cap \text{dom}(\sigma) = \emptyset$

Proof. Let $x \in X$ and $\sigma$ and $\tau$ substitutions that fulfill the requirements of the lemma.

$$(\sigma \triangleleft \tau)(x) = (\sigma \triangleleft (\tau(x))) = (\tau\sigma)|_{\text{dom}(\sigma)}(\tau(x))$$

Case $x \in \text{dom}(\sigma)$:

As $\text{dom}(\tau) \cap \text{dom}(\sigma) = \emptyset$ we know that $x \notin \text{dom}(\tau)$. Therefore

$$(\tau\sigma)|_{\text{dom}(\sigma)}(\tau(x)) = (\tau\sigma)|_{\text{dom}(\sigma)}(x) = (\tau\sigma)(x)$$

Case $x \notin \text{dom}(\sigma)$:

If also $x \notin \text{dom}(\tau)$ then $(\tau\sigma)(x) = x = ((\sigma \triangleleft \tau)(x))$. Otherwise $x \in \text{dom}(\tau)$. As $\text{var}(\text{rng}(\tau)) \cap \text{dom}(\sigma) = \emptyset$ we know from the definition of the restriction that

$$(\tau\sigma)|_{\text{dom}(\sigma)}(\tau(x)) = \tau(x)$$

But since $x \notin \text{dom}(\sigma)$ we also have $(\tau\sigma)(x) = \tau(\sigma(x)) = \tau(x)$. \qed

Lemma 2.4. If a given formula $\psi \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi$ is satisfiable, then the modified formula $\psi \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi \sigma$ is also satisfiable if there is a left-inverse $\xi^{-1}$ of $\xi$ such that:

1. $\text{dom}(\xi^{-1}) \cap \text{dom}(\sigma) = \emptyset$
2. $\text{dom}(\xi^{-1}) \cap \text{var}(\text{rng}(\sigma)) = \emptyset$
3. $\text{dom}(\xi^{-1}) \cap \text{var}(\psi) = \emptyset$

Proof. We have the following equivalences:

$$\psi \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi$$

$$\iff \psi \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi \xi^{-1} \sigma$$

$$\iff \psi \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi \xi^{-1} \sigma \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi \sigma$$

by Lemma 2.3 with 1. and 2.

$$\iff \psi(\xi^{-1} \land \sigma) \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi \xi^{-1} \sigma \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi \sigma$$

with 3.

$$\iff (\psi \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi \xi^{-1} \sigma)$$

Together with Lemma 2.2 it follows that also $\psi \land \bigwedge_{i=1}^{n} \varphi_i \sigma_i^\xi$ is satisfiable. \qed
2.3 SAT Modulo Theories

In this thesis we make use of SMT solvers for reasoning about first-order formulas. Therefore we describe the idea behind SMT solving and its main advantages. We use [13] as our main source for this description.

2.3.1 What is SMT?

Several tasks in software verification and analysis require to automatically determine the satisfiability of first-order formulas. However, normally one is not interested in the general satisfiability, i.e. satisfiability in any $\Sigma$-algebra with any interpretation of the symbols of the signature $\Sigma$. Instead, one wants to determine the satisfiability in some theory. For example, in a formula in the language of the theory of linear arithmetic, symbols like $<$ or $+$ should have the normal meaning.

Satisfiability Modulo Theories (SMT) refers to the problem of deciding the satisfiability of formulas in one or more theories. For many of those theories there exist specialized decision procedures that may exploit the mathematical properties of the theories involved. Therefore, those procedures are much more efficient in determining the satisfiability than for example general-purpose theorem provers, even if the latter are equipped with the axioms that define a theory.

In the last years, the intensified research on SMT solving techniques has led to powerful solvers that can handle formulas in the language of different theories and even combinations of theories. One reason for the strength of those solvers is that they make use of the advances in the field of SAT solving. For example SMT solvers build upon the ability of SAT solvers to handle propositional formulas with great efficiency.

2.3.2 The Theory behind SMT-Solving

At the moment, there are two main approaches to SMT solving. The first one, called *eager*, is based on encoding first-order formulas in pure propositional logic. The resulting propositional formula can then be handled with a SAT solver. The main problem with the eager approach is to find a suitable encoding. Depending on the background theory one has to expect exponential blow-ups in terms of formula size in the worst case.

The second approach, which is the one that is used by most of the implementations that competed in the annual SMT-competition[2], is called *lazy*. In its most simple form, the idea here is to use a SAT solver to enumerate satisfying assignments of the atoms of the formula and to use specialized decision procedures to check if these assignments are consistent with the background theory. Practical implementations of course do not adhere to this simple scheme but make use of numerous enhancement. For example
they may exploit the fact that the theory solvers may deduce the values of other atoms of the formula that are entailed by the current assignment and thus contribute to a faster termination of the algorithm. Similar, there are special enhancements to almost all techniques that play a role in modern SAT solvers like preprocessing, conflict analysis, etc.

**Combination of Theories**

As programs normally support many data types, each with their own theory, many real world problems involve reasoning in combinations of theories. Thus, in order to make use of SMT solvers for practical purposes in program analysis, they must be capable of handling more than only one theory at once. A simple example may be the theory of arrays combined with the theory of linear integer arithmetic.

Most current SMT solvers today implement theory combination by using a variant of the method based on Nelson and Oppen [41]. The idea behind this method is to separate a $\varphi$ into a conjunction of formulas $\varphi_1 \land \varphi_2$ where $\varphi_i$ is a $(\Sigma_i \cup C)$-formula. Here, $\Sigma_1$ and $\Sigma_2$ are disjoint signatures and $C$ is a set of constants not in $\Sigma_1 \cup \Sigma_2$. This form is called *pure*, as each sub-formula only contains symbols from one signature as well as constants from $C$. The process of converting a formula into a pure form is called *purification*.

With a formula in a pure form, $\varphi_1$ and $\varphi_2$ can be processed by a specialized solver. The problem here is that it does not suffice if one or even both solvers determine that $\varphi_1$ respectively $\varphi_2$ are satisfiable. To ensure that also the conjunction is satisfiable the two solvers must agree on an equivalence relation on the constants in $C$. Even this however, in general, does only work for certain kinds of theories. Furthermore, procedures for combining theories are rather expensive and thus many extensions and enhancements are applied in practice to keep the effort as low as possible. As the handling of equalities is an important task, several solvers have an architecture that consists of a core solver that handles equalities and satellite solvers that cover the other theories.

**Handling of Quantifiers**

Besides the built-in theories, one often needs to define an own theory by giving axioms of that theory. Those axioms, in turn, may use the built-in theories. Normally, one would define some (uninterpreted) sort together with some function symbols and give axioms that describe the functions. For example, later we will define a theory for memories and pointers. This involves axiomatizing functions for pointer arithmetic. Those axioms normally make use of quantifiers, especially universal quantifiers. However, for many theories, only the quantifier-free fragment is decidable. An example
for such a theory is the theory of uninterpreted functions. Even though the full theory may be undecidable, most SMT solvers implement an incomplete, heuristic-based decision procedure for theories with quantifiers. The main idea behind most quantifier handling is to use skolemization to eliminate existential quantifiers and to instantiate universally quantified formulae with terms. As the instantiation is in general non-exhaustive, it can only prove unsatisfiability. The main challenge with such a quantifier handling is to come up with a good heuristic to guide the instantiation. Some solvers allow the user to influence the instantiation by providing patterns from which the solver generates the instances.

Models and Unsatisfiable Subformulas

If a formula is satisfiable, SMT solvers are often able to produce a model of the formula, i.e. a satisfying assignment. On the other hand, if a formula is unsatisfiable many solvers can produce an unsatisfiable subformula, often called unsatisfiable core. One is often interested in small or even minimal unsatisfiable cores. However, normally the solvers do not guarantee the unsatisfiable cores to be minimal.

2.3.3 Summary: What SMT Solvers Can Do and Why

As we have seen, SMT solvers based on the lazy approach are normally not monolithic systems but instead build on a wide range of different techniques: a SAT solver at the core surrounded by specialized theory solvers. The main benefit of SMT solvers is the efficient combination of decision procedures for different fragments of first-order logic. Therefore, it is possible to model problems in a quite expressive language while still keeping the benefit of obtaining a solution fast. Regarding programming languages, it is possible to reason about many language constructs like arrays or recursive data types and therefore about a large set of properties of concrete programs. Furthermore, one may reason about hardware specific issues like the bit representation of integers as many SMT solvers provide decision procedures for a theory of bit vectors.
Chapter 3

False-Positive Elimination in Goanna

We now give some details on the current false-positive elimination procedure of Goanna. A comprehensive description can be found in [27]. We show the problems of this approach and then propose a new approach.

3.1 Interval based False-Positive Elimination

As explained in the last chapter, Goanna basically performs model checking on the control flow graph. For the false-positive elimination and model refinement steps, the CFG is labelled with interval equations that are derived from an interval semantics for C. Such an annotated CFG is called interval automaton. If the model checker returns a counter-example, it is interpreted as a system of interval equations. If this equation system has no least solution, the counter-example is spurious. It is now possible to reduce the equation system and to use the reduced system to build an observer model that is composed with the CFG to obtain a refined model. Figure 3.1 shows an example of a C-function and the interval automaton for that function. The main operations on intervals are $\cap$ and $\cup$, where $\cap$ is interval intersection and $\cup$ is interval union. Intersection is defined as usual and union is defined as

$$[l_1, u_1] \cup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$

3.2 Problems With the Current Approach

There are several problems with the current interval-based approach. The first one is the lack of a memory model. This prevents memory related operations like address calculation and pointer dereferencing from being adequately represented in the semantics. Partly, this problem can be solved by...
3.2 Problems With the Current Approach

void foo() {
    int x, *a;
    int* p = malloc(sizeof(int));
    for (x = 10; x > 0; x--) {
        a = p;
        if (x == 1)
            free(p);
    }
}

Figure 3.1: Example of an Interval Automaton for a function

using information from an aliasing analysis which identifies pointer variables that may or must alias. However, even with aliasing information, pointer arithmetic cannot be modelled adequately.

The second problem is the lack of precision that lies in the domain of intervals itself. Consider for example a code fragment like

void bar(int c) {
    if (c != 1) {...}
    ...
    if (c == 1) {...}
}

Assume that c is not modified in the code blocks that are left out. For some property, the model checker might return a counter-example that represents a path that enters both if blocks. The generated equation system contains the equations

\[ c_1 = [\infty; \infty] \]
\[ c_2 = (c_1 \cap [-\infty; 0]) \cup (c_1 \cap [2; \infty]) \]
\[ c_3 = c_2 \cap [1; 1] \]
The equation system has the following least solution:

\[ c_1 = [-\infty; \infty] \]
\[ c_2 = [-\infty; \infty] \]
\[ c_3 = [1; 1] \]

However, the path is clearly infeasible.

### 3.3 SMT based False-Positive Elimination

Due to the shortcomings of the approach based on solving interval equations the need emerged for a more precise false-positive elimination procedure. There are two main requirements for the new procedure. First, it must be able to handle a large fraction of the syntax and semantics of the C programming language and second it should do so on a level of abstraction that is precise enough to find typical false-positives but that still allows for a good performance. It is obvious that the procedure must also be fully automatic. In the following we outline an approach for a false-positive elimination that strives to fulfill the requirements stated above.

![Figure 3.2: Overview over the false-positive elimination cycle](image)

Initially, our approach is similar to the current interval-based one. We determine the feasibility of a counter-example path and then perform an iterative refinement. In order to be able to reason about the feasibility of counter-example paths, one needs to have a formalism to describe such paths. Therefore, we will develop a simple language for paths that uses a small set of statements and includes most of the syntax for C-expressions. To provide a semantics for this languages we have to take into account several issues. First, the domain used for the semantics must be one for which
3.3 SMT based False-Positive Elimination

there exist good automated reasoning procedures. Second, the mapping between syntax and semantic should not involve a big overhead. Third, it must be possible to find a good description of the memory in that language. This is important for handling pointer arithmetic. Because of the last issue, separation logic seems a good candidate, as it has been used in many successful verification projects and one of its aims is to enable reasoning about memory. However, the degree of automation for separation logic seems not high enough. We chose first-order logic as formalism for the semantics. As satisfiability in full first-order logic is not decidable and generic incomplete decision procedures do not seem to be fast enough, we only use a selection of first order theories together with a manually designed memory model. Due to recent developments in the field of SMT solvers, it is possible to decide reasonably sized formulas over these theories in a very short time. The range of theories that state-of-the-art SMT solvers support also seems broad enough to map a large portion of C into it.

The model-refinement step will be quite similar to Goanna’s current refinement based on the interval solver. Using an automata-based approach, the paths that violate the semantics of the path language will be iteratively eliminated from the program model.

Figure 3.2 gives an overview over the SMT based false-positive elimination cycle.
Chapter 4

A Language for Specifying Program Paths

This chapter introduces a language for describing program paths and gives an axiomatic semantics for this language.

4.1 Syntax

The syntax of the path language is made up of statements $s \in \text{Stmt}$ and expressions $e \in \text{Expr}$. A statement is one of assign, assume, call, eval, skip or a sequence of statements $s_1; s_2$. An expression may be every valid C-expression with a few exceptions. We do not handle compound literals and string literals\textsuperscript{1}. Furthermore, we currently do not support function pointers. The syntax for expressions in C can be found in the official C99-standard [33]. We make the simplifying assumption that identifiers (i.e. program variables) and field names are globally unique. We call a sequence of path statements a path program or sometimes just a path.

Assign

assign($e_1, e_2$) assigns the value of $e_2$ to $e_1$. However, not every expression may be used on the left hand side, but only modifiable l-values (see Section 4.2.4). On the right-hand side, every expression may be used.

Assume

assume($e$) checks the condition $e$ and fails if the condition does not hold in the current state.

\textsuperscript{1}In the implementation, we currently treat string literals as fresh pointer variables.
4.1 Syntax

Call

\( \text{call}(d, e_r, (e_1, \ldots, e_n)) \) denotes a call to the function designated by \( d \) with the values of \( e_1 \) to \( e_n \) as arguments and \( e_r \) as output parameter.

Eval

\( \text{eval}(e) \) evaluates an expression \( e \) for its side effects.

Skip

\( \text{skip} \) is the statement that does nothing.

Sequential Composition

\( s_1; s_2 \) denotes a sequential composition of two statements \( s_1 \) and \( s_2 \).

Comparison to C

Compared to C, the syntax of the path language is restricted in the sense that it only allows a small set of statements and strictly distinguishes between statements and expressions. One reason for this restriction is to control where side effects occur. The set of statements given here is sufficient to transform a path program into an equivalent one where no expressions with side effects occur. More details on this are given in section 4.2.1. A second reason why only a restricted set of statements is needed is that several control flow statements\(^2\) (like if, while, etc.) of the C language are not applicable to paths as they denote branches or jumps in the control flow. Those kinds of statements are replaced by assume if they represent a conditional branching. Otherwise they are discarded. For example there is no goto in the path language. Furthermore, there are no short-circuit operators, as they, too, result in branches.

In C, every expression is also a statement. Expressions are evaluated for their side effects. The counterpart for this kind of statement in our language is the eval statement. Although this would subsume function calls, there is a special statement for function calls (call).

Although the syntax for statements in our language is quite different to C, the syntax for expressions is mostly inherited from C. This means that certain constructs are redundantly defined, e.g. assignments or function calls. They may occur as a statement, but also as a sub-expression of a statement. When defining the semantics, this redundancy is removed in a transformation step where all expressions with potential side effects are

\(^2\)Referring to [33], this subsumes labeled-statements, selection-statements, iteration-statements and jump-statements.
replaced by expressions without side effects and statements that capture the side effects.

4.2 Semantics

This section describes the semantics of the path language. To do this, we first give a syntax transformation for path programs. This eliminates certain expressions and we only have to give the semantics for this reduced set of expressions. After that we develop a memory model, that formalizes memory-related concepts like pointers as well as the access and update of the memory. We will then describe the semantics in terms of this memory model.

4.2.1 Transformation

The idea of the transformation is to bring a statement into some kind of canonical form where redundancies and side effects are removed. We express side effects via new statements that we place in front of the original statement and replace expressions with equivalent side effect free versions. The transformation depends on the evaluation order of subexpressions. This, however, is compiler-specific. It would be possible to define a transformation that covers all possible evaluation orders by introducing a new variable that chooses between all orders. But this would result in a considerable blow-up of the program size. Here, we give a transformation for a specific evaluation order (right-to-left) which has to be adapted for the specific platform that the analysis targets. Related to the evaluation order is the question when side effects take place. The C standard imposes some restrictions on compilers by defining the concept of sequence points and demanding that side effects have to take place before the next sequence point. We assume here that side effects have taken place after the evaluation of an expression. This assumption may not be realistic for compilers that perform optimizations. But again, a general approach would result in a considerable blow-up of the complexity of the analysis.

During the transformation some further simplifications are applied: Compound assignments are replaced by simple assignments (e.g. $e_1 += e_2$ is replaced by $e_1 = e_1 + e_2$) and array accesses by pointer arithmetic (e.g. $e_1[e_2]$ is replaced by $*((e_1) + (e_2)))$. These replacements are consistent with the C99 standard. The operators which cause side effects are $=$, the postfix and prefix versions of $++$ and $--$, and function calls.

We formalize the transformation using a function $T_{stmt} : Stmt \rightarrow Stmt$ that transforms a statement $s$ into a statement $s'$, where no expression has side effects. $T_{stmt}$ also includes the simplifications mentioned above and removes the eval statement. $T_{stmt}$ makes use of the transformation function for expressions $T_{exp} : Exp \rightarrow Stmt \times Exp$, that takes an expression and
has a statement (possibly involving sequential composition) capturing the side effects of the expression and a transformed expression. Tables 4.1 and 4.2 show the definition of the two functions. For the rest of the chapter we only handle path programs that have been transformed in the way just described.

### 4.2.2 Memory Model

We now develop a theory of memories and pointers. The main part is the specification of the memory and the operations on it. We choose a rather simple and abstract model of the memory. We therefore discard certain details (e.g. there is no notion of padding in our model). An advantage of our model is that it does not need many axioms and thus allows efficient reasoning. On the other hand it is not an accurate model of the memory as seen from a C program and therefore incomplete.

#### Abstract Model

The model is based on the component-as-array-model that seems to have been first described by Burstall [15] and has been applied in various settings (e.g. [14, 28, 32, 39]). However, we adapt the model in order to allow for an unrestricted use of the address operator and for assignment of whole structures. The main feature of the component-as-array-model is the way it handles the access to components of structured objects. Instead of using the component as an index in a memory or an offset with regard to a base address, it has a dedicated memory (represented as array) for every field. If $o$ is some object and $f$ is some field, the access $o.f$ would be represented as $f[o]$. This has the big advantage that whenever the content of a field is altered we know that the content of every other field is not touched. This reflects the requirement of the C standard that structure fields occupy non-overlapping regions in the memory and therefore do not alias. In a memory-as-byte-sequence-model, where structure fields are typically represented by

<table>
<thead>
<tr>
<th>$s$</th>
<th>$T^{\text{stmt}}(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign($e_1, e_2$)</td>
<td>$s_2' ; s_1' ; \text{assign}(e_1', e_2')$</td>
</tr>
<tr>
<td>assume($e$)</td>
<td>$s_T' ; \text{assume}(e_T)$</td>
</tr>
<tr>
<td>call($d, e_r, e_1, \ldots, e_n$)</td>
<td>$s_T' ; \ldots; s_1' ; s_T' ; \text{call}(d, e_T', e_1', \ldots, e_2')$</td>
</tr>
<tr>
<td>eval($e$)</td>
<td>$s_T'$</td>
</tr>
<tr>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td>$s_1; s_2$</td>
<td>$T^{\text{stmt}}(s_1); T^{\text{stmt}}(s_2)$</td>
</tr>
</tbody>
</table>

where $(s_i', e_T') = T^{\text{exp}}(e_i)$.

Table 4.1: Transformation of statements
### 4.2 Semantics

<table>
<thead>
<tr>
<th>$e$</th>
<th>$\mathcal{T}^{\text{exp}}(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$id$</td>
<td>$(\text{skip}, id)$</td>
</tr>
<tr>
<td>$ic$</td>
<td>$(\text{skip}, ic)$</td>
</tr>
<tr>
<td>$fc$</td>
<td>$(\text{skip}, fc)$</td>
</tr>
<tr>
<td>$e_1[e_2]$</td>
<td>$\mathcal{T}^{\text{exp}}(e_1 + e_2)$</td>
</tr>
<tr>
<td>$d(e_1, \ldots, e_n)$</td>
<td>$(s^n_T; \ldots; s_1^T; \text{call}(d, id_{\text{new}}, e_1^T, \ldots, e_n^T), id_{\text{new}})$</td>
</tr>
<tr>
<td>$e.f$</td>
<td>$(s_T^T, e^T.f)$</td>
</tr>
<tr>
<td>$e \rightarrow f$</td>
<td>$(s_T^T, (e^T).f)$</td>
</tr>
<tr>
<td>$e + e$</td>
<td>$(s_T^T; \text{assign}(id_{\text{new}}, e_T^T); \text{assign}(e_T^T, e_T^T + 1), id_{\text{new}})$</td>
</tr>
<tr>
<td>$e - e$</td>
<td>$(s_T^T; \text{assign}(id_{\text{new}}, e_T^T); \text{assign}(e_T^T, e_T^T - 1), id_{\text{new}})$</td>
</tr>
<tr>
<td>$+ e$</td>
<td>$\mathcal{T}^{\text{exp}}(e = e + 1)$</td>
</tr>
<tr>
<td>$- e$</td>
<td>$\mathcal{T}^{\text{exp}}(e = e - 1)$</td>
</tr>
<tr>
<td>$\circ e$</td>
<td>$(s_T^T, \circ e_T^T)$</td>
</tr>
<tr>
<td>$(\text{type})e$</td>
<td>$(s_T^T, (\text{type})e_T^T)$</td>
</tr>
<tr>
<td>$\text{sizeof}(e)$</td>
<td>$(s_T^T, (\text{sizeof})e_T^T)$</td>
</tr>
<tr>
<td>$e_1 \triangleq e_2$</td>
<td>$(s_2^T; s_1^T; e_1^T \triangleq e_2^T)$</td>
</tr>
<tr>
<td>$e_1 \triangle e_2$</td>
<td>$\mathcal{T}^{\text{exp}}(e_1 = e_1 \triangle e_2)$</td>
</tr>
<tr>
<td>$e_1 = e_2$</td>
<td>$(s_1^T; s_2^T; \text{assign}(e_1^T, e_2^T), e_1^T)$</td>
</tr>
</tbody>
</table>

$\succeq \in \{+,-,\cdot,\slash,\%\}$,$\succ,<,\succ,\&,$，$\sim$,$\lnot$}

$\circ \in \{\&,\cdot,\succeq,\succ,\sim,\lnot\}$

$(s_i^T, e_i^T) = \mathcal{T}^{\text{exp}}(e_i)$

$id$ is an identifier expression

$ic$ ($fc$) is a constant integer (float) expression

$id_{\text{new}}$ is a fresh identifier with suitable type

| Table 4.2: Transformation of expressions |
offsets with regard to the base address of a data-object, the same fact would have to be asserted by axioms. See [35] for some details on this scheme. On the other hand, the idea that pointers are basically integers as it is expressed in the C standard doesn’t hold anymore for the component-as-array-model, which has to be considered in the semantics e.g. of addition.

In contrast to the original model, our memory model is segmented on several levels. First we use different memory variables for different data types. By using one distinct memory per type we additionally get the desirable property that there is no aliasing between expressions of different types\(^3\). On the second level, the logical memories are segmented into partitions. Every field occupies one partition in a logical memory. Every data object that is not the content of some field is stored in the distinguished partition Main. Following [37] we also introduce the notion of memory blocks. A memory block is a sequence of memory cells that is indexed by an offset. The block itself is associated with an address and has a certain size. Altogether the memory is made up of three hierarchies: memories, partitions and blocks and it thus takes three pieces of information to uniquely identify one memory cell. We abstract from the concrete bit representation of the data. The memory cells themselves may therefore store data of arbitrary size. We now formalize what we have described informally so far.

**The core model**  The core model uses several logical sorts. We assume we already have the sorts \(\mathbb{Z}\) and \(\mathbb{R}\) that are defined within the theories of integers and real numbers. Some of those sorts have constructors, which means that elements of that sort can be created from elements of other sorts. We will omit those constructors and use a set notation instead. The following definitions represent a signature. We will call this signature \(\Sigma_{\text{mem}}\) and will later assume a suitable set of variables \(X_{\text{mem}}\). All the axioms below are implicitly universally quantified.

\[
\begin{align*}
\text{id} & \in \text{Identifiers} \\
\text{f} & \in \text{Fields} \\
\text{a} & \in \text{Addresses} \\
\text{par} & \in \text{Partitions} = \{\text{Main}\} \cup \text{Fields} \\
p & \in \text{Pointers} = (\text{Partitions} \times \text{Addresses} \times \mathbb{Z}) \cup \{\text{Null}\} \\
v & \in \text{Values} = \mathbb{Z} \cup \mathbb{R} \cup \text{Pointers} \\
m & \in \text{Memories}
\end{align*}
\]

\(^3\)Here, our model takes a simplified view compared with C. In C, it is possible that pointers of different types point to the same location. Such a situation could be created by pointer casts. However, in most of the cases the assumption is legitimate.
4.2 Semantics

To access and update a memory we have two function symbols

\[ \text{access : Memories, Pointers} \rightarrow \text{Values} \]
\[ \text{update : Memories, Pointers, Values} \rightarrow \text{Memories} \]

Before we define the access and update functions, we introduce the predicate symbol \( \text{valid : Pointers} \). A pointer is valid if it is not \( \text{Null} \) and the offset is greater or equal than zero.

\[ \text{valid}(p) \iff p \neq \text{Null} \land (p = (\text{par}, a, o) \Rightarrow o \geq 0) \]

For convenience we write \( \text{valid}(p_1, \ldots, p_n) \) for \( \text{valid}(p_1) \land \ldots \land \text{valid}(p_n) \)

The update and access functions are defined by the following axioms:

**Axiom 1** (Access/Update 1).

\[ \text{valid}(p) \Rightarrow \text{access}(\text{update}(m, p, v), p) = v \]

**Axiom 2** (Access/Update 2).

\[ \text{valid}(p, p') \land p \neq p' \Rightarrow \text{access}(\text{update}(m, p, v), p') = \text{access}(m, p') \]

**Axiom 3** (Extension).

\[ m = m' \iff (\text{access}(m, p) = \text{access}(m', p)) \]

**Allocation Function**  The allocation function

\[ \text{VarAlloc : Identifiers} \rightarrow \text{Addresses} \]

assigns an address to each identifier.

**Pointer arithmetic**  We now define operations for pointer arithmetic. Pointer arithmetic is only allowed inside of blocks, which means that pointer arithmetic is arithmetic on the offset component of pointers. This restriction is due to the C standard that defines pointer arithmetic only inside of arrays.\(^4\) Other than in the C standard we only restrict the pointer arithmetic to not exceed the lower array bound (zero). The upper array bound is not enforced.\(^5\) There are three function symbols to denote addition between a

---

\(^4\)Actually, it is defined on the array bound plus one. Additionally, pointers that currently point beyond the array bounds may not be dereferenced. In effect, dereferencing pointers that were a result of pointer arithmetic never read the value of a data object outside the original array. This effect is achieved by restricting pointer arithmetic to blocks in our memory model.

\(^5\)The reason for this is that we do not support dynamic memory allocation. Therefore we generally cannot keep track of block sizes.
pointer and an integer, subtraction between a pointer and an integer and subtraction between two pointers respectively.

\[
padd : \text{Pointers}, \mathbb{N} \rightarrow \text{Pointers} \\
psub : \text{Pointers}, \mathbb{N} \rightarrow \text{Pointers} \\
ppsub : \text{Pointers}, \text{Pointers} \rightarrow \mathbb{N}
\]

They are axiomatized in the following way.

**Axiom 4** (Pointer addition).

\[
o + n \geq 0 \Rightarrow padd((par, a, o), n) = (par, a, o + n)
\]

**Axiom 5** (Pointer subtraction 1).

\[
o - n \geq 0 \Rightarrow psub((par, a, o), n) = (par, a, o - n)
\]

**Axiom 6** (Pointer subtraction 2).

\[
par_1 = par_2 \land a_1 = a_2 \Rightarrow ppsub((par_1, a_1, o_1) - (par_2, a_2, o_2))) = o_1 - o_2
\]

**Pointer Casts** C allows pointers to be cast to integers as well as to different types of pointers. Integers may as well be cast back to pointers. As our memory model, however, is not byte-precise, we cannot model those casts. Nevertheless we give a conservative approximation of these operations. We have three function symbols to model the casting:

\[
ipcast : \mathbb{Z} \rightarrow \text{Pointers} \\
opicast : \text{Pointers} \rightarrow \mathbb{Z} \\
ppcast : \text{Pointers} \rightarrow \text{Pointers}
\]

The operations are defined using the following axioms:

**Axiom 7** (Integer-to-Pointer Cast 1).

\[
ipcast(0) = \text{Null}
\]

**Axiom 8** (Integer-to-Pointer Cast 2).

\[
i \neq 0 \Rightarrow ipcast(i) \neq \text{Null}
\]

**Axiom 9** (Pointer-to-Integer Cast 1).

\[
opicast(\text{Null}) = 0
\]

**Axiom 10** (Pointer-to-Integer Cast 2).

\[
p \neq \text{Null} \Rightarrow picast(p) \neq 0
\]
4.2 Semantics

**Axiom 11** (Inverse 1).

\[ \text{ipcast}(\text{picast}(p)) = p \]

**Axiom 12** (Inverse 2).

\[ \text{picast}(\text{ipcast}(n)) = n \]

**Axiom 13** (Pointer-to-Pointer Cast 1).

\[ \text{ppcast}(\text{Null}) = \text{Null} \]

**Axiom 14** (Pointer-to-Pointer Cast 2).

\[ p \neq \text{Null} \Rightarrow \text{ppcast}(p) \neq \text{Null} \]

**Pointer comparison** As pointers may be compared to each other, we also need to define those comparisons. The C standard only defines comparisons between pointers that point to locations in the same array or aggregate object. We can easily determine if two pointers point to location in the same array. This is the case if the two block addresses are identical. However, in our model, we cannot in general determine if two pointer point to locations that are part of the same aggregate object (as the locations may represent objects on different levels of a nested data-structure). Therefore we also give an approximating axiomatization. We introduce special predicates for pointer comparison but only show the axiom for \( p \), the other cases are analogous.

**Axiom 15** (Pointer Less).

\[ a_1 = a_2 \land o_1 < o_2 \Rightarrow (\text{par}_1,a_1,o_1) <_p (\text{par}_2,a_2,o_2) \]

**Implementation Model**

In order to use an SMT solver to reason about the memory model, one has to map the concepts of the abstract model to something that the SMT solver can handle. This means we have to find an encoding of the memory model into the theories for which the solver has decision procedures.

As background theories we use the combination of the theories of linear real arithmetic, linear integer arithmetic, arrays and algebraic data types. The theory of algebraic datatypes is not a theory per se but rather a family of theories. The input language of an SMT solver typically allows for defining custom data type sorts.

**Fields, Partitions** and **Addresses** are represented by integers. Pointers are defined as an algebraic data type that has the two constructors **Pointer** and **Null**. **Memories** are represented by arrays. We have no analogon for **Values** but instead directly use the sorts integer and real or pointer.
Therefore we have one pair of access/update functions per sort and will write access\(_s\) and update\(_s\) to denote the access and update functions that operate on memories that store values of sort \(s\). We will now have a look at how the axioms of the abstract memory model have to be modified for the implementation model. As it turns out the only axioms that need to be modified are those that define access and update.

The theory of arrays with extensionality comes with two functions select and store that are axiomatized in the same way as Axioms (1-3), except for the valid clause in the antecedent of (1) and (2). Thus the axioms for functions for memory access and update can be formulated in the following way using arrays.

\[
\begin{align*}
\text{valid}(p) & \implies \text{access}_s(m,p) = \text{select}(m,p) \\
\text{valid}(p) & \implies \text{update}_s(m,p,v) = \text{store}(m,p,v)
\end{align*}
\]

The third axiom is not needed anymore as it is part of the theory of arrays. The axioms for the pointer related functions can be used without any change. For the rest of this chapter we use the abstract memory model, knowing how it can be implemented in a solver.

### 4.2.3 Type System

The data stored in memory may be of different types. We do not model the whole range of types that a language like C typically provides. Instead, we use just a few and map the C-types to them. Figure 4.1 shows the recursive definition of the types. Later we also give rules that define how to derive the type for an expression. We use the concepts of types and type environments as it can be found in [41].

\[
\begin{align*}
\tau & \in \text{Types} \\
\Gamma & \in \text{TypeEnvs}
\end{align*}
\]

#### Types

We have two arithmetic types \texttt{Int} and \texttt{Float}. Compound types are modeled as finite sequences of fields in \texttt{Fields} together with their types. Fields are globally unique, i.e. no two compound types share a field. We denote compound types as \texttt{Cmp}(\(f_1 : \tau_1, \ldots, f_n : \tau_n\)). For each type, there exists a corresponding pointer type. Arrays are not explicitly represented. Instead, an extended pointer type is used. Extended pointers are pairs containing the type pointed to and a length which is equal to zero for normal pointers and greater than zero for arrays. We write extended pointer types as \texttt{Ptr}(\tau, l) where \(\tau\) is a type and \(l\) is a natural number. Arithmetic types and pointer types are called scalar types.
4.2 Semantics

\[ \text{Types} = S \cup C \cup F \cup \{\text{Void}\} \]
\[ S = A \cup P \]
\[ A = \{\text{Int, Float}\} \]
\[ P = \{\text{Ptr}(\tau, l) \mid l \in \mathbb{N}\} \]
\[ C = \{\text{Cmp}(f_1 : \tau_1, \ldots, f_n : \tau_n) \mid j \neq k \Rightarrow f_j \neq f_k\} \]

Figure 4.1: The definition of the set of types

Type Environment

A type environment can be thought of as a partial function that maps an identifier to a type. We write \( \Gamma(id) \) for the type of the identifier \( id \) in the environment \( \Gamma \). The type environment mirrors the information which a programmer typically conveys using variable declarations.

Derivation Rules

The following rules define how to derive the type of an expression. A fact that in the type environment \( \Gamma \) the expression \( e \) is of type \( \tau \) is written as \( \Gamma \vdash e : \tau \). Before we give the derivation rules we define a shortcut that helps us to express when two types can be compared with the (un-)equality operators.

**Definition 4.1.** Two types are *comparable with respect to equality*, denoted by \( \tau_1 \sim \tau_2 \), if

- \( \tau_1, \tau_2 \in A \) or
- \( \tau_1 = \text{Ptr}(\tau_1', l_1), \tau_2 = \text{Ptr}(\tau_2', l_2) \) and
  - \( \tau_1' = \tau_2' \) or
  - \( \tau_1' = \text{Void} \) or
  - \( \tau_2' = \text{Void} \)
4.2 Semantics

(IntConst) \[ \Gamma \vdash ic : \text{Int} \]
(FloatConst) \[ \Gamma \vdash fc : \text{Float} \]

(IDENT) \[ \Gamma \vdash x : \tau \quad \text{if} \ \Gamma(x) = \tau \]

(DOT) \[ \Gamma \vdash e : \text{Cmp}(f_1 : \tau_1, \ldots, f_n : \tau_n) \]
\[ \Gamma \vdash e.f_i : \tau_i \]

(UnAddSub) \[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash oe : \tau \quad \circ \in \{+,-\} \quad \text{and} \quad \tau \in A \]

(BitComp) \[ \Gamma \vdash e : \text{Int} \]
\[ \Gamma \vdash \sim e : \text{Int} \]

(Not) \[ \Gamma \vdash e : \tau \quad \text{if} \ \tau \in S \]

(Deref) \[ \Gamma \vdash e : \text{Ptr}(\tau, l) \]
\[ \Gamma \vdash *e : \tau \]

(Addr) \[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash &e : \text{Ptr}(\tau, 0) \]

(Cast) \[ \Gamma \vdash e : \tau' \]
\[ \Gamma \vdash (\tau) e : \tau \]

(SizeOf) \[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash \text{sizeof}(e) : \text{Int} \]

(Arith 1) \[ \Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int} \]
\[ \Gamma \vdash e_1 \circ e_2 : \text{Int} \quad \circ \in \{+, -, *, /, \%\} \]

(Arith 2) \[ \Gamma \vdash e_1 : \text{Float} \quad \Gamma \vdash e_2 : \text{Float} \]
\[ \Gamma \vdash e_1 \circ e_2 : \text{Float} \quad \circ \in \{+, -, *, /\} \]

(Arith 3) \[ \Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Float} \]
\[ \Gamma \vdash e_1 \circ e_2 : \text{Float} \quad \circ \in \{+, -, *, /\} \]

(Arith 4) \[ \Gamma \vdash e_1 : \text{Float} \quad \Gamma \vdash e_2 : \text{Int} \]
\[ \Gamma \vdash e_1 \circ e_2 : \text{Float} \quad \circ \in \{+, -, *, /\} \]

(AddPtr 1) \[ \Gamma \vdash e_1 : \text{Ptr}(\tau, l) \quad \Gamma \vdash e_2 : \text{Int} \]
\[ \Gamma \vdash e_1 + e_2 : \text{Ptr}(\tau, 0) \]
Definition 4.2 (well typed expressions and statements). We call an expression $e$ well typed in a type environment $\Gamma$ if there is some $\tau$ such that $\Gamma \vdash e : \tau$.

A statement $s$ is well typed if

- $s = \text{skip}$ or
- $s = \text{assign}(e_1, e_2), \Gamma \vdash e_1 : \tau_1, \Gamma \vdash e_2 : \tau_2$ and $\tau_1 = \tau_2$ or
- $s = \text{assume}(e)$ and $\Gamma \vdash e : \tau$ and $\tau \in S$ or
- $s = \text{call}(d, e_r, e_1, \ldots, e_n)$ and $e_r$ and all $e_i$ are well typed or
- $s = \text{eval}(e)$ and $e$ is well typed or
- $s = s_1; \ldots; s_n$ and all $s_i$ are well typed.
4.2 Semantics

4.2.4 Semantics of Expressions

L-Values and R-Values

C distinguishes between two kinds of expressions, l-values and r-values. L-values are expressions that describe some object in the memory. The content of such a memory object may represent a value. For example, each variable is an l-value. An integer variable describes a memory region at a certain address with a certain length. The content of the memory represents an integer number. A special case of an l-value is a modifiable l-value. As the name suggests, it describes an object whose content may be modified. An r-value, on the opposite, does not describe a memory region but represents a value. For example, an integer constant is an r-value. An l-value may be converted to an r-value. On the other hand, an r-value may not be converted to an l-value.

Certain operators in C require one or more of its operands to be r-values or (modifiable) l-values. For example, the assignment operator requires its left operand to be a modifiable l-value and its right operand to be an r-value. In case the right operand is also an l-value, it is converted to an r-value. To reflect these issues in our semantics, we need two semantical domains, one for l-values and one for r-values. The main idea is to use terms over the memory model and some basic theories as semantic domain.

In our memory model, the information needed to uniquely specify where a data object is stored is the type of that object (as we have one memory variable per type) and a pointer. A pair containing these two pieces is called a semantic memory location. However, we must cater for the case when we cannot express a certain pointer in our background theory. Therefore, the pointer component in a location may also be ∅.

Let $\Sigma_{sem}$ be the signature of our combined theory (containing $\Sigma_{mem}$) and $X_{sem}$ a set of variables. We write $\mathcal{J}_s$ for $\mathcal{J}(\Sigma_{sem}, X_{sem})$. Semantic memory locations are formalized as follows.

$$sloc \in SLocations = \{ (\tau, p) \mid \tau \in Types, p \in \mathcal{T}Pointers \cup \{\top\} \}$$

As an aggregate object (a structure or array) may occupy several memory cells, it should also be represented by several locations. Therefore, we define an l-value recursively by

$$lval \in Lvals = (Lvals \cup SLocations)^*$$

The definition of r-values is similar to the definition of l-values but instead of semantic memory locations, r-values contain values. A value may either be a term denoting an element of Values or $\top$ if we cannot express a value in our theories.

$$sval \in SValues = \mathcal{T}Values \cup \{\top\}$$
The definition of r-values then has the same structure as the one for l-values.

\[ rv \in Rvals = (Rvals \cup SValues)^* \]

**Address Computation**

In the semantics of l-values we sometimes need to evaluate an expression for its address and offset. Therefore we define the address semantics of an expression.

\[ A[\cdot] : Expr \times TypeEnvs \rightarrow (\mathcal{T}_{\text{Address}} \times \mathcal{T}_z) \cup \{ \bot \} \]

It may return \( \tau \) in case the address and the offset cannot be determined and \( \perp \) if the expression is not well typed.

**Not well typed expressions**

\[ A[e] \Gamma = \perp \quad \text{if} \quad \exists \tau. \Gamma \vdash e : \tau \]

**Identifier**

\[ A[id] \Gamma = (VarAlloc(id), 0) \]

**Structure and Union Member**

\[ A[e.f] \Gamma = A[e] \Gamma \quad \text{if} \quad \Gamma \vdash e.f : \tau \in S \]
\[ A[e.f] \Gamma = \text{access}(\tau_{\text{Mem}}, (f, a, o)), 0) \quad \text{if} \quad \Gamma \vdash e.f : \tau \notin S \]
\[ \quad \text{and} \quad A[e] \Gamma = (a, o) \]
\[ A[e.f] \Gamma = \tau \quad \text{if} \quad \Gamma \vdash e.f : \tau \notin S \]
\[ \quad \text{and} \quad A[e] \Gamma = \tau \]

**Pointer Dereferencing**

\[ A[*e] \Gamma = (a, o) \quad \text{if} \quad R[e] \Gamma = (par, a, o) \]
\[ A[*e] \Gamma = \tau \quad \text{if} \quad R[e] \Gamma = \tau \]

**Semantics of L-Values**

The semantics of l-values is given by the function

\[ L[\cdot] : Expr \times TypeEnvs \rightarrow Lvals \cup \{ \perp \} \]

Like the address computation function it returns \( \perp \) for expressions that are not well typed. In order to handle the recursive structure of l-values we use the helper function **unfold** which is recursively defined as
4.2 Semantics

\[
\text{unfold}(\text{Ptr}(\tau, l), e, \Gamma) = (\text{unfold}(\tau, e + 0, \Gamma), \ldots, \\
\text{unfold}(\tau, e + (l - 1)), \Gamma) \quad \text{if } l > 0
\]

\[
\text{unfold}(\text{Cmp}(f_1 : \tau_1, \ldots, f_n : \tau_n), e, \Gamma) = (\text{unfold}(\tau_1, e.f_1, \Gamma), \ldots, \\
\text{unfold}(\tau_n, e.f_n, \Gamma))
\]

\[
\text{unfold}(\tau, e, \Gamma) = L[e] \Gamma
\]

Not well typed expressions

\[
L[e] \Gamma = \bot \quad \text{if } \exists \tau. \Gamma \vdash e : \tau
\]

Identifier

\[
L[id] \Gamma = (\tau, (\text{Main, VarAlloc(id)}, 0)) \quad \text{if } \Gamma \vdash id : \tau \in S
\]

\[
L[id] \Gamma = \text{unfold}(\tau, id, \Gamma) \quad \text{if } \Gamma \vdash id : \tau \notin S
\]

Structure and Union Member

\[
L[e.f] \Gamma = (\tau, (f, a, o)) \quad \text{if } \Gamma \vdash e.f : \tau \in S \text{ and } A[e.f] \Gamma = (a, o)
\]

\[
L[e.f] \Gamma = (\tau, \top) \quad \text{if } \Gamma \vdash e.f : \tau \in S \text{ and } A[e.f] \Gamma = \top
\]

\[
L[e.f] \Gamma = \text{unfold}(\tau, e.f, \Gamma) \quad \text{if } \Gamma \vdash e.f : \tau \notin S
\]

Pointer Dereferencing

\[
L[\ast e] \Gamma = (\tau, p) \quad \text{if } \Gamma \vdash \ast e : \tau \in S \text{ and } R[e] \Gamma = p \neq \top
\]

\[
L[\ast e] \Gamma = (\tau, \top) \quad \text{if } \Gamma \vdash \ast e : \tau \in S \text{ and } R[e] \Gamma = \top
\]

\[
L[\ast e] \Gamma = \text{unfold}(\tau, \ast e, \Gamma) \quad \text{if } \Gamma \vdash \ast e : \tau \notin S
\]

Semantics of R-Values

The semantics of r-values is given by the function

\[
\mathcal{R[\cdot]} : \text{Expr} \times \text{TypeEnvs} \to \text{Rvals} \cup \{\bot\}
\]

Like for the two semantic functions above, also the r-value semantics returns \(\bot\) for expressions that are not well typed.

The semantics for r-values is also defined for l-values. In such a case the C standard prescribes a l-value-to-r-value conversion. This conversion is done by accessing the memory at the locations given by the l-value. We define a shortcut, lval-to-rval for this conversion. It is defined as

\[
lval-to-rval((\tau, \top)) = \top
\]

\[
lval-to-rval((\tau, p)) = \text{access}(\tau_{\text{Mem}}, p) \quad \text{if } p \neq \top
\]

\[
lval-to-rval(lv_1, \ldots, lv_n) = (lval-to-rval(lv_1), \ldots, lval-to-rval(lv_n))
\]
A special case is the semantics for arrays. If an array expression is used as r-value, a pointer value is returned instead of the content of the array. There is a special rule for array-typed expressions, for all rules afterwards we assume the type of the expression is not an array (i.e. an extended pointer type with \( l \neq 0 \)).

In the following, we allow if-then-else-terms (ite-terms) and write them as \( \varphi \ ? \ t_1 \ \triangleright \ t_2 \). Such a term evaluates to \( t_1 \) if \( \varphi \) is true and otherwise to \( t_2 \). This informal extension to the logic is legitimate as we are interested in satisfiability of formulas and for a formula that contains ite-terms there is an equisatisfiable formula without ite-terms [34].

For ease of notation let \( v = \mathcal{R}[e] \Gamma \) and \( v_i = \mathcal{R}[e_i] \Gamma \).

**Not well typed expressions**

\[
\mathcal{R}[e] \Gamma = \perp \quad \text{if } \not\exists \tau. \Gamma \vdash e : \tau
\]

**Array Expressions**

\[
\mathcal{R}[\text{e.f}] \Gamma = (f, a, o) \quad \text{if } \Gamma \vdash \text{e.f} : \text{Ptr}(\tau, l) \text{ with } l > 0 \quad \text{and } \mathcal{A}[\text{e.f}] \Gamma = (a, o)
\]

\[
\mathcal{R}[\text{e.f}] \Gamma = \top \quad \text{if } \Gamma \vdash \text{e.f} : \text{Ptr}(\tau, l) \text{ with } l > 0 \quad \text{and } \mathcal{A}[\text{e.f}] \Gamma = \top
\]

\[
\mathcal{R}[\text{e}] \Gamma = (\text{Main}, a, o) \quad \text{if } \Gamma \vdash \text{e} : \text{Ptr}(\tau, l) \text{ with } l > 0 \quad \text{and } \mathcal{A}[\text{e}] \Gamma = (a, o)
\]

\[
\mathcal{R}[\text{e}] \Gamma = \top \quad \text{if } \Gamma \vdash \text{e} : \text{Ptr}(\tau, l) \text{ with } l > 0 \quad \text{and } \mathcal{A}[\text{e}] \Gamma = \top
\]

From now on we assume that the expressions in double brackets are not of array type and we do not write this explicitly as a constraint.

**Constants**

\[
\mathcal{R}[ic] \Gamma = ic
\]

\[
\mathcal{R}[fc] \Gamma = fc
\]

**Identifier**

\[
\mathcal{R}[id] \Gamma = \text{lval-to-rval}(\mathcal{L}[id] \Gamma)
\]
4.2 Semantics

Structure and Union Member
\[ \mathcal{R}[e.f] \Gamma = \text{val-to-rval} (\mathcal{L}[e.f] \Gamma) \]

Pointer Dereferencing
\[ \mathcal{R}[e] \Gamma = \text{val-to-rval} (\mathcal{L}[e] \Gamma) \]

Unary Expressions
\[ \mathcal{R}[+e] \Gamma = 0 + v \quad \text{if } v \neq \top \]
\[ \mathcal{R}[+e] \Gamma = \top \quad \text{if } v = \top \]
\[ \mathcal{R}[-e] \Gamma = 0 - v \quad \text{if } v \neq \top \]
\[ \mathcal{R}[-e] \Gamma = \top \quad \text{if } v = \top \]
\[ \mathcal{R}[-\sim e] \Gamma = \text{bitcomp}(v) \quad \text{if } v \neq \top \]
\[ \mathcal{R}[-\sim e] \Gamma = \top \quad \text{if } v = \top \]
\[ \mathcal{R}![e] \Gamma = v \rightarrow 0 ? 1 \triangleright 0 \quad \text{if } v \neq \top \text{ and } \Gamma \vdash e : \tau \in A \]
\[ \mathcal{R}![e] \Gamma = \text{picast}(v) = 0 ? 1 \triangleright 0 \quad \text{if } v \neq \top \text{ and } \Gamma \vdash e : \tau \in P \]
\[ \mathcal{R}![e] \Gamma = \top \quad \text{if } v = \top \]

Address Operator
\[ \mathcal{R}[&(e.f)] \Gamma = (f, a, o) \quad \text{if } \mathcal{A}[e.f] \Gamma = (a, o) \]
\[ \mathcal{R}[&(e.f)] \Gamma = \top \quad \text{if } \mathcal{A}[e.f] \Gamma = \top \]
\[ \mathcal{R}[&(*e)] \Gamma = \mathcal{R}[e] \Gamma \]
\[ \mathcal{R}[&e] \Gamma = (\text{Main}, a, o) \quad \text{if } \mathcal{A}[e] \Gamma = (a, o) \]
\[ \mathcal{R}[&e] \Gamma = \top \quad \text{if } \mathcal{A}[e] \Gamma = \top \]

Sizeof Operator
\[ \mathcal{R}[sizeof(e)] \Gamma = \top \]
4.2 Semantics

**Cast Expression**

\[ \mathcal{R}[(\tau)(e)] \Gamma = v \quad \text{if } \Gamma \vdash e : \tau' \text{ and } \tau = \tau' \notin P \]

\[ \mathcal{R}[(\tau)(e)] \Gamma = \text{picast}(v) \quad \text{if } \tau = \text{Int} \text{ and } \Gamma \vdash e : \tau' \in P \text{ and } v \neq \top \]

\[ \mathcal{R}[(\tau)(e)] \Gamma = \text{ipcast}(v) \quad \text{if } \Gamma \vdash e : \text{Int} \text{ and } \tau \in P \text{ and } v \neq \top \]

\[ \mathcal{R}[(\tau)(e)] \Gamma = \text{ppcast}(v) \quad \text{if } \Gamma \vdash e : \tau' \text{ and } \tau, \tau' \in P \text{ and } v \neq \top \]

**Multiplication**

As we use linear arithmetic as background theory, we only define the semantics of multiplication expressions for linear expressions. Furthermore we do not define multiplication for mixed types.

\[ \mathcal{R}[e_1 \times e_2] \Gamma = v_1 \times v_2 \quad \text{if } \Gamma \vdash e_1 : \tau \text{ and } \Gamma \vdash e_2 : \tau' \quad \text{and } \tau = \tau' \in A \quad \text{and } e_1 \times e_2 \text{ is linear and } v_1, v_2 \neq \top \]

\[ \mathcal{R}[e_1 \times e_2] \Gamma = \top \quad \text{otherwise} \]

**Division**

As we use linear arithmetic as background theory, we only define the semantics for division expressions if the divisor is a constant. As with multiplication expressions we do not define the case of mixed types. Note that div means integer division, “/” means real division.

\[ \mathcal{R}[e / ic] \Gamma = v \div (\mathcal{R}[ic] \Gamma) \quad \text{if } \Gamma \vdash e : \text{Int} \text{ and } v \neq \top \]

\[ \mathcal{R}[e / ic] \Gamma = \top \quad \text{otherwise} \]

\[ \mathcal{R}[e / fc] \Gamma = v / (\mathcal{R}[fc] \Gamma) \quad \text{if } \Gamma \vdash e : \text{Float} \text{ and } v \neq \top \]

\[ \mathcal{R}[e / fc] \Gamma = \top \quad \text{otherwise} \]

\[ \mathcal{R}[e_1 / e_2] \Gamma = \top \]

**Modulo**

As with division expressions, the semantics of modulo expressions is only defined if the second operand is a constant. Furthermore, modulo is only defined for integer operands.

\[ \mathcal{R}[e \% ic] \Gamma = v \% (\mathcal{R}[ic] \Gamma) \quad \text{if } \Gamma \vdash e : \text{Int} \text{ and } v \neq \top \]

\[ \mathcal{R}[e \% ic] \Gamma = \top \quad \text{otherwise} \]

\[ \mathcal{R}[e_1 \% e_2] \Gamma = \top \]
### 4.2 Semantics

#### Addition

The operands in addition expressions may be of arithmetic type as well as of pointer type. However, the semantics for these cases is different. Again we do not define the semantics for mixed types.

\[
\begin{align*}
\mathcal{R}[e_1 + e_2] \Gamma &= v_1 + v_2 & \text{if } \Gamma \vdash e_1 : \tau \text{ and } \Gamma \vdash e_2 : \tau' \\
& \quad \text{and } \tau = \tau' \in A \text{ and } v_1, v_2 \neq \top \\
\mathcal{R}[e_1 + e_2] \Gamma &= \text{padd}(v_1, v_2) & \text{if } \Gamma \vdash e_1 : \tau \in P \text{ and } \Gamma \vdash e_2 : \text{Int} \\
& \quad \text{and } v_1, v_2 \neq \top \\
\mathcal{R}[e_1 + e_2] \Gamma &= \text{padd}(v_2, v_1) & \text{if } \Gamma \vdash e_2 : \tau \in P \text{ and } \Gamma \vdash e_1 : \text{Int} \\
& \quad \text{and } v_1, v_2 \neq \top \\
\mathcal{R}[e_1 + e_2] \Gamma &= \top & \text{otherwise}
\end{align*}
\]

#### Subtraction

The operands in subtraction expressions may be of arithmetic type. Furthermore one may subtract an integer from a pointer and subtract two pointers of the same type.

\[
\begin{align*}
\mathcal{R}[e_1 - e_2] \Gamma &= v_1 - v_2 & \text{if } \Gamma \vdash e_1 : \tau \text{ and } \Gamma \vdash e_2 : \tau' \\
& \quad \text{and } \tau = \tau' \in A \text{ and } v_1, v_2 \neq \top \\
\mathcal{R}[e_1 - e_2] \Gamma &= \text{psub}(v_1, v_2) & \text{if } \Gamma \vdash e_1 : \tau \in P \\
& \quad \text{and } \Gamma \vdash e_2 : \text{Int} \text{ and } v_1, v_2 \neq \top \\
\mathcal{R}[e_1 - e_2] \Gamma &= \text{ppsub}(v_1, v_2) & \text{if } \Gamma \vdash e_1 : \tau \text{ and } \Gamma \vdash e_1 : \tau' \\
& \quad \text{and } \tau = \tau' \in P \text{ and } v_1, v_2 \neq \top \\
\mathcal{R}[e_1 - e_2] \Gamma &= \top & \text{otherwise}
\end{align*}
\]

#### Bitwise Operators

We use uninterpreted functions to model the effects of bitwise operators.

\[
\begin{align*}
\mathcal{R}[e_1 >> e_2] \Gamma &= \text{shiftr}(v_1, v_2) & \text{if } \Gamma \vdash e_1 : \text{Int} \text{ and } \Gamma \vdash e_2 : \text{Int} \\
& \quad \text{and } v_1, v_2 \neq \top \\
\mathcal{R}[e_1 >> e_2] \Gamma &= \top & \text{otherwise} \\
\mathcal{R}[e_1 << e_2] \Gamma &= \text{shiftl}(v_1, v_2) & \text{if } \Gamma \vdash e_1 : \text{Int} \text{ and } \Gamma \vdash e_2 : \text{Int} \\
& \quad \text{and } v_1, v_2 \neq \top \\
\mathcal{R}[e_1 << e_2] \Gamma &= \top & \text{otherwise}
\end{align*}
\]
4.2 Semantics

\[ R[e_1 \& e_2] \Gamma = band(v_1, v_2) \quad \text{if } \Gamma \vdash e_1 : \text{Int} \text{ and } \Gamma \vdash e_2 : \text{Int} \text{ and } v_1, v_2 \neq \top \]

\[ R[e_1 \& e_2] \Gamma = \tau \quad \text{otherwise} \]

\[ R[e_1 | e_2] \Gamma = bor(v_1, v_2) \quad \text{if } \Gamma \vdash e_1 : \text{Int} \text{ and } \Gamma \vdash e_2 : \text{Int} \text{ and } v_1, v_2 \neq \top \]

\[ R[e_1 | e_2] \Gamma = \tau \quad \text{otherwise} \]

\[ R[e_1 ^ e_2] \Gamma = bxor(v_1, v_2) \quad \text{if } \Gamma \vdash e_1 : \text{Int} \text{ and } \Gamma \vdash e_2 : \text{Int} \text{ and } v_1, v_2 \neq \top \]

\[ R[e_1 ^ e_2] \Gamma = \tau \quad \text{otherwise} \]

**Relational Operators**

The C Standard only defines comparison between two expressions of arithmetic type or two expressions of pointer-type. Again, because we do not handle mixed arithmetic we only define the semantics for arithmetic types if both expressions have the same type. For the inequalities we only write down the case “\(\leq\)”. The other cases are analogous.

\[ R[e_1 \leq e_2] \Gamma = v_1 \leq v_2 ? 1 : 0 \quad \text{if } \Gamma \vdash e_1 : \tau \text{ and } \Gamma \vdash e_2 : \tau' \text{ and } \tau = \tau' \in A \text{ and } v_1, v_2 \neq \top \]

\[ R[e_1 \leq e_2] \Gamma = v_1 \leq_p v_2 ? 1 : 0 \quad \text{if } \Gamma \vdash e_1 : \tau \text{ and } \Gamma \vdash e_2 : \tau' \text{ and } \tau = \tau' \in P \text{ and } v_1, v_2 \neq \top \]

\[ R[e_1 \leq e_2] \Gamma = \tau \quad \text{otherwise} \]

**Equality Operators**

\[ R[e_1 = e_2] \Gamma = (v_1 = v_2) ? 1 : 0 \quad \text{if } \Gamma \vdash e_1 : \tau \text{ and } \Gamma \vdash e_2 : \tau' \text{ and } \tau = \tau' \in S \text{ and } v_1, v_2 \neq \top \]

\[ R[e_1 = e_2] \Gamma = \tau \quad \text{otherwise} \]

**Example 4.1**

To illustrate the semantics, we apply it on an example. We want to calculate \(R[o[1].f] \Gamma\) where \(\Gamma(o) = \text{Ptr}(f : \text{Int}, 2)\). For the examples we assume a memory state as depicted in Figure 4.2. We would normally not apply the
4.2 Semantics

4.2.4 Semantics of Statements

We now give the semantics of statements, based on the semantics of expressions that we have just defined. The semantics of statements is an axiomatic semantics that describes statements in terms of weakest preconditions. In the classical definition of weakest preconditions, program variables are syntactically substituted by other terms. However, this is not applicable in a context where aliasing can occur. Therefore, we adopt a slightly different approach. Instead of substituting program variables, we substitute memory variables with memory updates.
4.2 Semantics

Weakest Preconditions

Weakest preconditions were originally proposed by Dijkstra [22] as a way of specifying a semantics for his language of guarded commands. The idea of weakest preconditions is, given a postcondition for some statement, to compute a constraint on the program state such that the postcondition holds after executing the statement in a state that fulfills the constraint. However, as the name suggests, the computed constraint is the weakest of such constraints. The classical definition of the weakest precondition is given as syntactical transformations of the postcondition. E.g. for an assignment the definition of \( wp \) is:

\[
wp(x := a, \varphi) = \varphi\{x \mapsto a\}
\]

However, this definition of \( wp \) is no longer valid when \( x \) may be an alias of some other expression. For example in the statement \( x[i] := a \), we might not only need to substitute \( x[i] \) but also some \( x[j] \) in the case when \( i \) and \( j \) have the same value. A solution for the array case is to substitute the whole array by an updated array.

\[
wp(x[i] := a, \varphi) = \varphi\{x \mapsto x[i \mapsto a]\}
\]

This approach can be generalized to accommodate every type of assignment. Every assignment can be seen (and is indeed treated by the semantics)
as a modification of some memory. The memory plays now the part of the array above. So instead of substituting program variables we substitute memory variables by memory updates. At this point the memory model comes into focus. As the l-values describe exactly which memory is being modified by an assignment and no aliasing can occur across memory borders, we can just substitute the memory variable used for the memory described by the l-value. As we use one memory variable per type, we need a way to map types to the memory variable. Let therefore $\tau_{Mem}$ denote the memory variable used for type $\tau$. So given a type environment $\Gamma$ and given that $L[e] \Gamma = (\tau, p)$, the weakest precondition for an assignment $e : = a$ is

$$wp(e : = a, \varphi) = \varphi\{\tau_{Mem} \mapsto \text{update}(\tau_{Mem}, p, a)\}$$

One of the advantages of the classical weakest precondition semantics is that it enables a local reasoning. Only those parts of a formula are substituted that may be affected by the statement. By switching to a memory based formulation of the weakest precondition we have to give up some of this locality. At this point it becomes clear, why we split up the memory in several parts: to have at least some locality. In the cited works about the component-as-array-model it is argued that by using one memory per field the locality is increased even more. However, we would have to give up address computation for fields. The following small path program illustrates the problem:

assign(p,&x.f);
assign(*p,5);
assume(x.f!=5)

The problem occurs when calculating the weakest precondition of the second statement. The pointer $p$ does not carry any information that it points somewhere into $f$’s memory. Therefore we would have to update all memories mentioned before (i.e. speaking in bottom to top direction). In the example this would be only $f$ but in general it would be every memory that stores data of the type, $p$ points to. Essentially this is what is done with the current model, except that there is only one memory per type anyway. We could run an analysis beforehand in order to determine for each pointer to which memory it may point and thus use a model with greater locality.

Now we give the complete semantics for statements, based on the ideas outlined so far.

**wp-transformer for statements**

$wp$ is a function that takes a path statement $s$ and a post-condition $\varphi$ and returns the weakest precondition of $s$ with respect to $\varphi$. We define $wp(s, \varphi)$ in a way such that the result always has the form $\varphi \sigma \land \psi$ where $\sigma$ is a
substitution. This may seem a little cumbersome at first but it makes it possible to assume some kind of normal form for the weakest precondition of an arbitrary path. As the weakest precondition of course takes recourse to the semantics of expressions, the \( wp \) transformer is parametrized with a type environment \( \Gamma \). However, later we will omit this parameter if it is clear from the context which type environment is meant. The next sections present the definition of \( wp^\Gamma(s, \varphi) \). We will assume that \( s \) is well typed. This means that also every sub-statement and every sub-expression of \( s \) is well typed. Therefore we do not need to handle the case where an expression semantics returns \( \bot \).

**Assign**

Other than in traditional definitions of \( wp \), the effect of an assignment in our definition is not just represented by a substitution in the postcondition. Instead we encode the fact that the memory has changed with a substitution of a memory variable and then constrain that change by a formula. Without going into details here, the main reason for this is that we can later establish a relationship between subformulas of weakest preconditions and infeasible sub-paths.

We define a helper function \( assign(lv, rv) \) that takes an l-value and an r-value and returns a variable substitution and a formula. Five cases have to be distinguished, depending on \( lv \) and \( rv \). If the r-value is \( \tau \) we do not constrain the value that is written to the memory. The symbols \( \theta \) and \( \Delta \) stand for fresh variables.

Case \( lv = (\tau, p), p \neq \tau \) and \( rv \neq \tau \):

\[
assign(lv, rv) = (\{\tau_{Mem} \mapsto update(\tau_{Mem}, p, \theta)\}, \theta = rv)
\]

Case \( lv = (\tau, p), p \neq \tau \) and \( rv = \tau \):

\[
assign(lv, rv) = (\{\tau_{Mem} \mapsto update(\tau_{Mem}, p, \theta)\}, \text{true})
\]

Case \( lv = (\tau, \tau) \) and \( rv \neq \tau \):

\[
assign(lv, rv) = (\{\tau_{Mem} \mapsto update(\tau_{Mem}, \Delta, \theta)\}, \theta = rv)
\]

Case \( lv = (\tau, \tau) \) and \( rv = \tau \)

\[
assign(lv, rv) = (\{\tau_{Mem} \mapsto update(\tau_{Mem}, \Delta, \theta)\}, \text{true})
\]

Case \( lv = (lv_1, \ldots, lv_n) \) and \( rv = (rv_1, \ldots, rv_n) \):

\[
assign(lv, rv) = (\sigma_1 \ldots \sigma_n, \land \psi_i) \quad \text{with} \quad (\sigma_i, \psi_i) = assign(lv_i, rv_i)
\]
Now we define
\[
wp^\Gamma(\text{assign}(e_1, e_2), \varphi) = \varphi \land \psi \quad \text{with} \quad (\sigma, \psi) = \text{assign}(L[e_1] \Gamma, R[e_2] \Gamma)
\]

\textbf{Assume}

Case \(R[e] \Gamma = \top\)
\[
wp^\Gamma(\text{assume}(e), \varphi) = \varphi \land \text{true}
\]

Case \(R[e] \Gamma = v \neq \top \text{ and } \Gamma \vdash e : \text{Int}\)
\[
wp^\Gamma(\text{assume}(e), \varphi)) = \varphi \land v \neq 0
\]

Case \(R[e] \Gamma = v \neq \top \text{ and } \Gamma \vdash e : \tau \in P\)
\[
wp^\Gamma(\text{assume}(e), \varphi) = \varphi \land \text{picast}(v) \neq 0
\]

\textbf{Call}
The weakest precondition of a function call needs some discussion. The context of this semantics is an intraprocedural one. This means there are no information about the exact behaviour of the called function, especially concerning the side effects of the function. However, we assume we have a knowledge about what a function may change. In [40] Morgan introduces the specification statement for such a situation. The idea is to specify a function by pre- and postconditions (which may be true) and by the collection of variables that may be changed during the execution of the function body. Although we cannot use this formalism directly as it is defined for the traditional approach to weakest preconditions, our solution using \textit{function summaries} is based on the same idea. A function summary is a function that takes a sequence of expressions \(e\) and returns a sequence of l-values. The idea is that given the actual parameters for the function call, the function summary gives the l-values that may be changed by the execution of the function. Let \(S_d\) denote the function summary for a function designator \(d\). Then we can express the weakest precondition of a function call by assigning \(\top\) to all the l-values that the summary returns. Furthermore, as we do not know anything about the return value either, we also assign \(\top\) to \(e_r\)

\[
wp^\Gamma(\text{call}(d, e_r, e_1, \ldots, e_n), \varphi) = \varphi(\sigma_2 \sigma_1) \land (\psi_1 \land \psi_2)
\]

where \((\sigma_1, \psi_1) = \text{assign}(S_d(e_1, \ldots, e_n), (\top, \ldots, \top))\)
and \((\sigma_2, \psi_2) = \text{assign}(e_r, \top)\)
4.2 Semantics

Eval

\[ wp^\Gamma(\text{eval}(e), \varphi) = \varphi \sigma_\varnothing \land \text{true} \]

Skip

\[ wp^\Gamma(\text{skip}(e), \varphi) = \varphi \sigma_\varnothing \land \text{true} \]

Sequence

\[ wp^\Gamma(s_1; s_2, \varphi)) = wp^\Gamma(s_1, wp(s_2, \varphi)) \]

Definition 4.3 (Infeasible Paths). A path \( p \) is called infeasible in a type environment \( \Gamma \) if \( wp^\Gamma(p, \text{true}) \) is unsatisfiable. An equivalent condition is that \( wp^\Gamma(p, \text{true}) \Rightarrow \text{false} \) is valid. We will omit \( \Gamma \) and just call a path infeasible if it is clear from the context which type environment we refer to.

4.2.6 Properties of \( wp \)

We will now prove some properties of \( wp \) that we will use in later chapters.

**Lemma 4.1.** For every statement \( s \) there is a formula \( \varphi \) and a substitution \( \sigma \) such that for every formula \( \psi \)

\[ wp(s, \psi) \iff \varphi \land \psi \sigma \]

**Proof.** We give a proof by structural induction over the structure of \( s \).

For assign, assume, call, eval and skip the property is obviously fulfilled as the definition of \( wp \) explicitly specifies \( \sigma \) and \( \varphi \). The only case that is left is the case of a sequence \( s_1; s_2 \). We have the following equivalences:

\[
\begin{align*}
wp(s_1; s_2, \psi) &= wp(s_1, wp(s_2, \psi)) \\
&\iff wp(s_1, \varphi_2 \land \psi \sigma_2) \quad \text{Induction Hypothesis} \\
&\iff \varphi_1 \land (\varphi_2 \land \psi \sigma_2) \sigma_1 \quad \text{Induction Hypothesis} \\
&\iff \varphi_1 \land \varphi_2 \sigma_1 \land \psi \sigma_2 \sigma_1 \\
\end{align*}
\]

Setting \( \varphi = \varphi_1 \land \varphi_2 \sigma_1 \) and \( \sigma = \sigma_2 \sigma_1 \), the last formula has the desired form.

**Lemma 4.2** (Excluded Miracle). For all \( s \), \( wp(s, \text{false}) \iff \text{false} \)

**Proof.** \( wp(s, \text{false}) \iff \varphi \land \text{false} \) for some \( \varphi \) by Lemma 4.1. However, \( \varphi \land \text{false} \iff \text{false} \)
Lemma 4.3 (Monotonicity of wp). If \( \varphi \Rightarrow \chi \) is valid, then for all statements \( s \) also \( wp(s, \varphi) \Rightarrow wp(s, \chi) \) is valid.

Proof. Given \( \varphi \) and \( \chi \) such that \( \varphi \Rightarrow \chi \) is valid. Because of Lemma 4.1 there is a formula \( \psi \) and a substitution \( \sigma \) such that \( wp(s, \varphi) \Rightarrow wp(s, \chi) \) is equivalent to \( \varphi \sigma \land \psi \Rightarrow \chi \sigma \land \psi \). As, obviously, \( (\varphi \Rightarrow \chi) \sigma \) is valid, so is \( \varphi \sigma \Rightarrow \chi \sigma \) and \( \varphi \sigma \land \psi \Rightarrow \chi \sigma \land \psi \).

Lemma 4.4. Every path with an infeasible sub-path is also infeasible.

Proof. Let \( p = p_1; q; p_2 \) be a path that has the infeasible sub-path \( q \), i.e. \( wp(q, true) \Rightarrow false \). As \( wp(p_2, true) \Rightarrow true \) and \( wp \) is monotone

\[ wp(q; p_2, true) = wp(q, wp(p_2, true) \Rightarrow wp(q, true) \Rightarrow false \]

The lemma of the excluded miracle now gives us

\[ wp(p_1; q; p_2, true) = wp(p_1, wp(q; p_2, true)) \Rightarrow false \]

Definition 4.4 (Contribution Normal Form). Let \( s_1; \ldots; s_n \) be a sequence of path statements, \( \varphi_1, \ldots, \varphi_{n+1} \) formulae and \( \sigma_0, \ldots, \sigma_n \) variable substitutions. Let furthermore

\[ \varphi_{cf} = \bigwedge_{i=1}^{n+1} \varphi_i \sigma_{i-1} \ldots \sigma_0 \]

We say that \( \varphi_{cf} \) is in \textit{contribution normal form} (CoNF) for \( s_1; \ldots; s_n \) if

- \( \sigma_0 = \sigma_0 \) and
- \( \varphi_{n+1} = true \)
- \( wp(s_i, \psi) \iff \varphi_i \land \psi \sigma_i \) for \( 1 \leq i \leq n \) and all \( \psi \)

If a formula \( \varphi \) is in CoNF for a path \( p \), every statement of \( p \) is related to a conjunct of \( \varphi \). We show that \( \varphi \) is furthermore equivalent to the weakest precondition of \( p \) with respect to \( true \). The following lemma formulates this claim.

Lemma 4.5. Given a path \( p \) and a formula \( \varphi_{cf} \) that is in CoNF for \( p \). Then \( wp(p, true) \iff \varphi_{cf} \).

Proof. We proof the proposition by induction over the length \( l \) of the path.

\( l = 1 \): Let \( p = s_1 \) and \( \varphi_{cf} = \varphi_1 \sigma_0 \land \varphi_2 \sigma_1 \sigma_0 \) be in CoNF for \( s_1 \). Because of the definition of CoNF, \( \sigma_0 = \sigma_0 \) and \( \varphi_2 = true \). This means that \( \varphi_{cf} \iff \varphi_1 \). Furthermore it holds that \( wp(s_1, \psi) \iff \varphi_1 \land \psi \sigma_1 \) for all \( \psi \). For the special
case $\psi = \text{true}$ we have $wp(s_1,\text{true}) \iff \varphi \iff \varphi_{cf}$ which is the proposition.

$l - 1 \rightarrow l$:
Let $p = s_1; \ldots; s_l$ and $\varphi_{cf} = \land_{i=1}^{l+1} \varphi_i \sigma_{i-1} \ldots \sigma_0$ be in CoNF for $p$. We know that $wp(s_1; \ldots; s_l,\text{true}) = wp(s_1, wp(s_2; \ldots; s_l,\text{true}))$ because of the definition of $wp$ for sequences. As $\varphi_{cf}$ is in CoNF for $p$, we have

$$wp(s_1,\psi) \iff \varphi \land \psi \sigma_i \text{ for all } \psi \text{ for } 1 \leq i \leq l$$

We introduce a path $p' = s'_1; \ldots; s'_{l-1}$, formulae $\varphi'_1, \ldots, \varphi'_{l-1}$ and substitutions $\sigma'_0, \ldots, \sigma'_{l-1}$ such that

- $s'_i = s_{i+1}$ for $1 \leq i \leq l - 1$
- $\varphi'_i = \varphi_{i+1}$ for $1 \leq i \leq l$
- $\sigma'_0 = \sigma_0$
- $\sigma'_i = \sigma_{i+1}$ for $1 \leq i \leq l - 1$

Now, $\varphi'_{cf} = \land_{i=1}^{l+1} \varphi'_i \sigma'_{i-1} \ldots \sigma'_0$ is in CoNF for $s'_1; \ldots; s'_{l-1}$. As $s'_1; \ldots; s'_{l-1}$ is a sequence of length $l - 1$ we can apply the induction hypothesis. This justifies the following equivalences.

$$wp(p,\text{true})$$
$$= wp(s_1, wp(s_2; \ldots; s_l,\text{true}))$$
$$\iff wp(s_1, wp(s'_1; \ldots; s'_{l-1},\text{true}))$$
$$\iff wp(s_1, \varphi'_{cf})$$

**Induction Hypothesis**

$$\iff \varphi_1 \land \varphi'_{cf} \sigma_1$$

$\varphi_{cf}$ is in CoNF for $p$

$$\iff \varphi_1 \land \land_{i=1}^{l+1} \varphi'_i \sigma'_{i-1} \ldots \sigma'_0 \sigma_1$$

$\sigma'_0 = \sigma_0$

$$\iff \varphi_1 \land \land_{i=1}^{l} \varphi'_i \sigma'_{i-1} \ldots \sigma'_1 \sigma_1$$

$\sigma'_0 = \sigma_0$

$$\iff \varphi_1 \land \land_{i=1}^{l} \varphi_{i+1} \sigma_i \ldots \sigma_2 \sigma_1$$

$$\iff \varphi_1 \land \land_{i=2}^{l} \varphi_i \sigma_{i-1} \ldots \sigma_2 \sigma_1$$

$$\iff \varphi_1 \sigma_0 \land \land_{i=2}^{l+1} \varphi_i \sigma_{i-1} \ldots \sigma_2 \sigma_1 \sigma_0$$

$\sigma_0 = \sigma_0$

$$\iff \land_{i=1}^{l+1} \varphi_i \sigma_{i-1} \ldots \sigma_0$$

$\square$
Lemma 4.6. For every path $s_1; \ldots ; s_n$ there are formulae $\varphi_1, \ldots , \varphi_{n+1}$ and substitutions $\sigma_0, \ldots , \sigma_n$ such that $\Lambda_{i=1}^{n+1} \varphi_i \sigma_{i-1} \ldots \sigma_0$ is in CoNF for $s_1; \ldots ; s_n$.

Proof. For $1 \leq i \leq n$ we know from Lemma 4.1 that there are suitable $\varphi_i$ and $\sigma_i$. We can now freely set $\varphi_{n+1} = true$ and $\sigma_0 = \sigma_\emptyset$. \qed
Chapter 5

Path Reduction

In this Chapter, we develop a formal representation of programs (labelled control flow graphs) using the path language. Based on this we use the framework of non-deterministic automata to characterize feasible paths in a program. Furthermore we present an abstraction of this formalism and show how to refine this abstraction in case of a spurious error. We use the basic ideas for automata-based refinement from [27] but adapt it for the use of our path language instead of interval equations.

5.1 Finite and Infinite Automata

As the presentations in this chapter uses some simple results from automata theory we give the notation and basic ideas. For a comprehensive description, see [31].

Definition 5.1 (Automata). A non-deterministic automaton is represented by a tuple

\[ A = (S, I, A, R, F) \]

where:

- \( S \) is a set of states.
- \( I \subseteq S \) is a set of initial states.
- \( A \) is an alphabet.
- \( R \subseteq S \times A \times S \) is a transition relation that relates an input state, a symbol of \( A \) and an output state.
- \( F \) is a set of final states.

An automaton is called \textit{finite-state} if its set of states \( S \) is finite, otherwise it is called \textit{infinite-state}.
5.2 Representing Programs as Automata

**Definition 5.2** (Execution and Language of Automata). An *execution* of an automaton $P = (S, I, A, R, F)$ is a finite sequence $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} s_n$ where $s_i \in S$, $a_i \in A$, $s_0 \in I$ and $(s_i, a_i, s_{i+1}) \in R$. An execution is accepting if $s_n \in F$. We say $w = a_0 \ldots a_{n-1} \in A^*$ is a word accepted by $P$ if there is an accepting execution $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_n} s_n$. The *language* of $P$, denoted by $\mathcal{L}(P)$, is the set of words that are accepted by $P$.

**Definition 5.3** (Product Automaton). The *product* $P_1 \times P_2$ of two automata $P_1 = (S_1, I_1, A_1, R_1, F_1)$ and $P_2 = (S_2, I_2, A_2, R_2, F_2)$ is the automaton represented by $(S_1 \times S_2, I_1 \times I_2, A_1 \times A_2, R_1 \times R_2, F_1 \times F_2)$, where $((s_1, s_2), a, (s'_1, s'_2)) \in R_1 \times R_2$ if $(s_1, a, s'_1) \in R_1$ and $(s_2, a, s'_2) \in R_2$.

The product of two finite-state automata is well defined as it is again a finite-state automaton. The reason for this is that the set of states of the product automaton is the cartesian product of the sets of states of the factors. The cartesian product of two finite sets, however, is finite.

**Lemma 5.1.** The language of the product automaton $P_1 \times P_2$ is the intersection of the languages of $P_1$ and $P_2$.

*Proof.* Let $P_1 = (S_1, I_1, A_1, R_1, F_1)$, $P_2 = (S_2, I_2, A_2, R_2, F_2)$ and $P_\times = P_1 \times P_2$. Let further $w = a_0 \ldots a_n \in A^*$.

We first show that if $w$ is in $\mathcal{L}(P_\times)$ then it is also in $\mathcal{L}(P_1) \cap \mathcal{L}(P_2)$. As $w \in \mathcal{L}(P_\times)$, there are $(q_0, p_0), \ldots, (q_{n+1}, p_{n+1})$ such that the execution $(q_0, p_0) \xrightarrow{a_0} \ldots \xrightarrow{a_n} (q_{n+1}, p_{n+1})$ is an accepting execution of $P_\times$. Then, $q_0 \in I_1$ and $q_{n+1} \in F_1$ and $(q_i, p_i, q_{i+1}, p_{i+1}) \in R_1 \times R_2$ for $0 \leq i \leq n$. Therefore, the execution $q_0 \xrightarrow{a_0} \ldots \xrightarrow{a_n} q_{n+1}$ is an accepting execution of $P_1$ and $w \in \mathcal{L}(P_1)$. With the same argumentation we can show that $w \in \mathcal{L}(P_2)$, which means that $w \in \mathcal{L}(P_1) \cap \mathcal{L}(P_2)$.

Now we show that if $w \in \mathcal{L}(P_1) \cap \mathcal{L}(P_2)$, then $w \in \mathcal{L}(P_\times)$. Because $w$ is in the intersection, $w \in \mathcal{L}(P_1)$ and $w \in \mathcal{L}(P_2)$. Then there must be $q_0, \ldots, q_{n+1}$ and $p_0, \ldots, p_{n+1}$ such that $q_0 \xrightarrow{a_0} \ldots \xrightarrow{a_n} q_{n+1}$ is an accepting execution of $P_1$ and $p_0 \xrightarrow{a_0} \ldots \xrightarrow{a_n} p_{n+1}$ is an accepting execution of $P_2$. The pairs $(q_i, p_i)$ are states of the product automaton. Furthermore, $(q_0, p_0)$ is an initial state of $P_\times$, as $q_0 \in I_1$ and $p_0 \in I_2$. As well, $(q_{n+1}, p_{n+1})$ is an accepting state of $P_\times$. As $(q_i, p_i, q_{i+1}, p_{i+1}) \in R_1 \times R_2$, it holds that $((q_i, p_i), a_i, (q_{i+1}, p_{i+1})) \in R_\times$. Thus, $(q_0, p_0) \xrightarrow{a_0} \ldots \xrightarrow{a_n} (q_{n+1}, p_{n+1})$ is an accepting execution of $P_\times$ and, because of that, $w \in P_\times$.

### 5.2 Representing Programs as Automata

We introduce the concept of *labelled control flow graphs* (LCFG) in order to represent programs. As the name suggests, an LCFG is similar to a control flow graph of a program. The difference is that the edges are labelled with
5.2 Representing Programs as Automata

statements of the path language. An LCFG accepts a language that we will define via an automaton. The accepted language of that automaton will be the set of all feasible (sub-)paths through the program with regard to the semantics outlined in the previous chapter.

Definition 5.4 (Labelled Conrol Flow Graph). A labelled control flow graph (LCFG) is represented by a triple \((L, E, stmt)\) where

- \(L\) is a finite set of locations
- \(E \subseteq L \times L\) is a finite set of edges
- \(stmt : E \rightarrow Stmt\) is a labelling function that assigns a path statement to each edge

Definition 5.5 (Semantic LCFG Automaton). Given a type environment \(\Gamma\) and an LCFG \(P = (L, E, stmt)\), the semantic LCFG automaton \(Sem(P, \Gamma)\) for \(P\) is the automaton \((S, S_0, E, R, F)\) where:

- \(S\) is the set of states \((l, \varphi)\) with location \(l \in L\) and state constraint \(\varphi \in \mathcal{F}(\Sigma_{sem}, X_{sem})\).
- \(S_0\) exactly contains all states of the form \((l, \varphi)\), such that \(\varphi \not\Rightarrow false\).
- \(R \subseteq S \times E \times S\) is a transition relation such that \(((l, \varphi), (l, l'), (l', \varphi'))\) is in the relation if \(\varphi = wp(stmt(l, l'), \varphi')\).
- \(F = \{(l, true) \mid l \in L\}\).

Words of \(Sem(P, \Gamma)\) are sequences of edges of \(P\), like \((l_0, l_1), \ldots, (l_{n-1}, l_n)\). In the following we will identify such a sequence with \(l_0, \ldots, l_n\).

Definition 5.6. The language \(\mathcal{L}(P)\) of a LCFG \(P\) with regard to the type environment \(\Gamma\) is the language of its semantic path automaton \(\mathcal{L}(Sem(P, \Gamma))\).

We will now show the characteristic property of LCFGs, namely that their language represents exactly the feasible paths through their underlying CFG with regard to some type environment. Before we can prove this in Proposition 5.1, we need some auxiliary lemmas.

As there is at most one edge between two states in a semantic LCFG automaton, we may for reasons of brevity identify an execution with the sequence of its states. Furthermore we will allow sequences of locations in \(wp\). For example, instead of \(wp(stmt(l_0, l_1); \ldots; stmt(l_{n-1}, l_n), \varphi)\), we may write \(wp(l_0 \ldots l_n, \varphi)\) if it is clear from the context which labelling function we refer to.
Lemma 5.2. Let $P$ be an LCFG and $\Gamma$ a type environment. If
\[ e = (l_0, \varphi_0) \ldots (l_{n-1}, \varphi_{n-1})(l_n, true) \]
is an accepting execution of $\text{Sem}(P, \Gamma)$ then also
\[ e' = (l_1, \varphi_1) \ldots (l_{n-1}, \varphi_{n-1})(l_n, true) \]
is an accepting execution of $\text{Sem}(P, \Gamma)$.

Proof. Obviously, $e'$ respects the transition relation and the last state of $e'$
is a final state. We need to show that $(l_1, \varphi_1)$ is an initial state, i.e. that
$\varphi_1 \not\rightarrow \text{false}$. Suppose this was false. Then $\varphi_0 = \text{wp}(\text{stmt}(l_1, l_2), \varphi_1) \Rightarrow \text{false}$
because of the excluded-miracle property of $\text{wp}$. Therefore, $(l_0, \varphi_0)$ is not
initial. This however is a contradiction to the assumption. $\square$

Lemma 5.3. Let $P$ be an LCFG and $\Gamma$ a type environment. If
\[ (l_0, \varphi_0) \ldots (l_{n-1}, \varphi_{n-1})(l_n, true) \]
is an accepting execution of $\text{Sem}(P, \Gamma)$, then $\varphi_0 = \text{wp}(l_0 \ldots l_n, true)$

Proof. Proof by induction over the length $k$ of the execution.

$k = 2$:
Let $e = (l_0, \varphi_0)(l_1, true)$ be an accepting execution of length 2. In this case
the proposition follows directly from the definition of the transition relation.

$k \rightarrow k + 1$:
Let $e = (l_0, \varphi_0) \ldots (l_{k-1}\varphi_{k-1})(l_k, true)$ be an accepting execution of length
$k + 1$. By Lemma 5.2, the subsequence $(l_1, \varphi_1) \ldots (l_{k-1}, \varphi_{k-1})(l_k, true)$ is an
accepting execution of length $k$. We apply the induction hypothesis and
have
\[ \varphi_1 = \text{wp}(l_1 \ldots l_k, true) \]
As $((l_0, \varphi_0), (l_0, l_1), (l_1, \varphi_1))$ is in the transition relation it follows that
\[ \varphi_0 = \text{wp}(l_0l_1, \varphi_1) \]
\[ = \text{wp}(l_0l_1, \text{wp}(l_1 \ldots l_k, true)) \]
\[ = \text{wp}(l_0 \ldots l_k, true) \]
$\square$

Proposition 5.1. Let $P = (L, E, \text{stmt})$ be an LCFG. A word $w = l_0 \ldots l_n$ is
in the language of $P$ with regard to the type environment $\Gamma$ iff
\[ \cdot (l_{i-1}, l_i) \in E \text{ for } 0 < i \leq n \text{ and } \]

5.3 Initial Abstraction and Refinement Loop

- \( s = \text{stmt}(l_0,l_1); \ldots; \text{stmt}(l_{n-1},l_n) \) is a feasible path with regard to \( \Gamma \).

Proof. “\( \Rightarrow \)”: Let \( w = l_0 \ldots l_n \) be a word in \( \mathcal{L}(\text{Sem}(P,\Gamma)) \). The first property is obviously fulfilled by the construction of the semantic LCFG automaton. Because \( w \) is a word in \( \mathcal{L}(\text{Sem}(P,\Gamma)) \), it follows that there must exist \( \varphi_0, \ldots, \varphi_{n-1} \) such that \( (l_0,\varphi_0) \ldots (l_{n-1},\varphi_{n-1})(l_n,\text{true}) \) is an accepting execution of \( \text{Sem}(P,\Gamma) \). Furthermore, \( \varphi_0 \not\Rightarrow \text{false} \) as \( \varphi_0 \) is part of the initial state. From Lemma 5.3 we now get that \( \varphi_0 = \text{wp}(s,\text{true}) \). Therefore, the path denoted by \( s \) is feasible.

“\( \Leftarrow \)”: We have a word \( w = l_0 \ldots l_n \) that fulfills the two properties. For each \( i \) with \( 0 \leq i \leq n-1 \) we set \( \varphi_i = \text{wp}(l_i \ldots l_n,\text{true}) \). This construction ensures that \( \varphi_i = \text{wp}(\text{stmt}(l_i,l_{i+1}),\varphi_{i+1}) \). As the path described by \( w \) is feasible, it follows that \( \varphi_0 \not\Rightarrow \text{false} \). As \( (l_i,l_{i+1}) \in E \) and because of the construction of \( \varphi_i \), the execution \( (l_0,\varphi_0) \ldots (l_{n-1},\varphi_{n-1})(l_n,\text{true}) \) is an accepting execution of \( \text{Sem}(P,\Gamma) \) and thus \( w \) is a word of \( P \).

Proposition 5.1 shows that an LCFG captures all possible behaviours of a program.

5.3 Initial Abstraction and Refinement Loop

As a semantic LCFG automaton for an LCFG is not finite – which makes it unsuitable for model checking – we use a finite abstraction of it. The language of such an abstraction is a superset of the language of the LCFG. Thus it is possible that the analysis yields spurious counter-examples. If that happens we use the counter-example to compute a refined abstraction and start the analysis again. This analysis-refinement-cycle will give rise to a sequence of automata that approximate the semantic LCFG automaton with increasing precision. We will now formalize abstractions and after that show how to derive an abstraction from an LCFG.

**Definition 5.7.** An automaton \( A' \) is an abstraction of an automaton \( A \) if \( \mathcal{L}(A) \subseteq \mathcal{L}(A') \).

**Definition 5.8.** A abstract semantic LCFG automaton for \( P \) is an automaton that is an abstraction of the semantic LCFG automaton for \( P \).

We assume we start our procedure with an LCFG \( P \). The first abstract semantic LCFG automaton for \( P \) is constructed directly from \( P \). We call this automaton the initial abstraction. It basically represents the (finite) control structure of the LCFG.

**Definition 5.9.** Given a type environment \( \Gamma \) and an LCFG \( P = (L,E,\text{stmt}) \), the initial abstraction \( \hat{P}_0 \) of \( P \) is the automaton \( (L,L,E,T,L) \) where the transition relation \( T = \{((l,(l,l'),l')) | (l,l') \in E\} \). All states in \( L \) are initial states and final states.
\( \hat{P}_0 \) is an abstraction of \( \text{Sem}(P, \Gamma) \), as obviously each word of \( P \) is also a word of \( \hat{P}_0 \). Therefore \( \hat{P}_0 \) is an abstract semantic LCFG automaton for \( P \). Furthermore \( \hat{P} \) is finite.

As the initial abstraction is finite we can use it for model checking. Through iterative checking and refinement – the latter still needs to be explained, which we will do in the next section – we produce a series of abstract semantic LCFG automata \( \hat{P}_0, \ldots, \hat{P}_n \). We will not give details on how the model checking is done but we assume that it produces a counter-example if some property is violated. We interpret a counter-example that emerged in the \( i \)-th iteration as a word that is in \( \mathcal{L}(\hat{P}_i) \). If this counter-example is spurious, that means it is not in \( \mathcal{L}(P) \). In this case we compute an observer automaton \( \text{Obs}_i \). This observer is constructed in a way that by combining it with \( \hat{P}_i \) we are guaranteed to obtain a new, refined abstraction that does not show the particular spurious behaviour any more. This is our new abstract semantic LCFG automaton \( \hat{P}_{i+1} \) which we can use for the next iteration. In the next sections we will show how we can detect spurious counter-examples and how we can compute the mentioned observers.

### 5.4 Building Observers From Infeasible Cores

We now describe how we build observer automata from spurious words.

**Definition 5.10 (Projection).** A path \( q = s'_1; \ldots; s'_k \) is a projection of a path \( p = s_1; \ldots; s_n \), if \( s'_1 \ldots s'_k \) is a sub-sequence of \( s_1; \ldots; s_n \).

**Definition 5.11 (Infeasible Core).** A path \( p' \) is an infeasible core of a path \( p \) if the following holds:

- \( p' \) is a projection of \( p \) and
- \( wp(p', \text{true}) \Rightarrow \text{false} \) and
- every projection \( p'' \) of \( p \), that has in turn \( p' \) as a projection, is infeasible.

We will use the terms projection and infeasible core also for words over locations of LCFGs, as they represent paths.

Intuitively, infeasible cores formalize what we called explanations above. It resembles very much the notion of unsatisfiable subformulas and indeed we will show later that we can use unsatisfiable subformulas of weakest preconditions to find infeasible cores of paths. At this point, we assume that there is a procedure to check if a word is spurious and if so calculate an infeasible core. The details of such a procedure will be delivered in the next section.

We now show how we build an observer for a word \( w \) that is spurious with regard to an LCFG \( P \). The aim is to exclude at least that word but
5.4 Building Observers From Infeasible Cores

potentially more from the current abstract semantic LCFG automaton from which \( w \) was generated. Similar as in [27], the observer automaton is an automaton such that it accepts \( w \) and that every word it accepts is not in \( L(P) \). This relationship is formalized in the following definition.

**Definition 5.12.** An automaton \( P_w \) separates the word \( w \) from an LCFG \( P \) if \( w \in L(P_w) \) and \( L(P_w) \cap L(P) = \emptyset \).

The automaton for the next iteration is then computed by building the product of the current abstract semantic LCFG automaton and the complement of the observer. The following lemma shows that such a product construction is correct in the sense that it maintains the abstraction relation and eliminates \( w \).

**Lemma 5.4.** Let \( \Gamma \) be a type environment, \( P \) an LCFG and \( \hat{P} \) an abstract semantic LCFG automaton for \( P \) with regard to \( \Gamma \). Let further \( w \) be a word such that \( w \in L(\hat{P}) \) and \( w \notin L(P) \). If an automaton \( \text{Obs} \) separates \( w \) from \( P \) then \( \hat{P} \times \text{Obs}^c \) is still an abstract semantic LCFG automaton for \( P \) and its language does not contain \( w \).

**Proof.** The argument is based on the fact that the language of the product of two automata is the intersection of their languages. We first show that \( \hat{P} \times \text{Obs}^c \) is an abstraction of \( \text{Sem}(P, \Gamma) \). Because of the mentioned fact about the language of product automata we know that \( L(\hat{P} \times \text{Obs}^c) = L(\hat{P}) \cap L(\text{Obs}^c) \). As \( L(\text{Obs}) \cap L(\text{Sem}(P, \Gamma)) = \emptyset \) it follows that \( L(\text{Obs}^c) \supseteq L(\text{Sem}(P, \Gamma)) \). Together with \( L(\hat{P}) \supseteq L(\text{Sem}(P, \Gamma)) \) we can now conclude that \( L(\hat{P} \times \text{Obs}^c) = L(\hat{P}) \cap L(\text{Obs}^c) \supseteq L(\text{Sem}(P, \Gamma)) \).

Furthermore, \( w \) cannot be in the language of the product automaton as it is not in \( L(\text{Obs}^c) \).

We now give a detailed description of the construction of an observer for a spurious word. The observer construction is simpler than in [27] but does not exclude as many potential spurious words. Briefly, if we have found a spurious word \( w \) and an infeasible core \( w' \), we build an observer automaton that accepts all words that contain the hull \( h \) of \( w' \) in \( w \) as sub-path.

**Definition 5.13** (Hull). Given a word \( w = l_1 \ldots l_n \) and an infeasible core \( w' = l_{i_0} \ldots l_{i_k} \) of \( w \), we call \( l_{i_0}l_{i_0+1} \ldots l_{i_k} \) the hull of \( w' \) in \( w \).

**Definition 5.14** (LCFG Observer Automaton). Let \( w \) be a word of an LCFG with edge-set \( E \), \( w' \) an infeasible core of \( w \) and \( h = l_{i_0} \ldots l_{i_k} \) the hull of \( w' \) in \( w \). The LCFG observer automaton \( \text{Obs}(E, w, w') \) is the automaton \((S_{\text{Obs}}, S_0, E, T, F)\), where

- \( S_{\text{Obs}} \) is the set of states \( \{s_1, \ldots, s_{k-1}\} \cup \{\text{Init}, \text{Conflict}\} \).
5.4 Building Observers From Infeasible Cores 54

- \( S_0 = \{ \text{init} \} \) is the set of initial states.
- \( T \subseteq S_{\text{Obs}} \times E \times S_{\text{Obs}} \) is the transition relation. A triple \((s, (l, l'), s')\) is in the relation if and only if one of the following holds
  1. \( s = \text{init} \) and \( s' = \text{init} \) and \((l, l') \neq (l_0, l_1)\)
  2. \( s = \text{Conflict} \) and \( s' = s_1 \) and \((l, l') = (l_0, l_1)\)
  3. \( s = s_i \) and \( s' = s_{i+1} \) and \((l, l') = (l_i, l_{i+1})\), \(1 \leq i \leq k - 2\)
  4. \( s = s_i \) and \( s' = \text{init} \) and \((l, l') \in E \setminus \{(l_0, l_1), (l_i, l_{i+1})\}\), \(1 \leq i \leq k - 1\)
  5. \( s = s_{k-1} \) and \( s' = \text{Conflict} \) and \((l, l') = (l_{k-1}, l_k)\)
  6. \( s = \text{Conflict} \) and \( s' = \text{Conflict} \)
- \( F = \{ \text{Conflict} \} \) is the set of final states.

We will now show that an LCFG observer automaton constructed as above separates \( w \) from \( P \). To do this, we will first prove some auxilary lemmas.

**Lemma 5.5.** Let \( w \) be a word over alphabet \( E \), \( w' \) an infeasible core of \( w \) and \( h \) the hull of \( w' \) in \( w \). \( \text{Obs}(E, w, w') \) accepts exactly the words over \( E \) that contain \( h \) as sub-path.

**Proof.** Let \( h = l_0 \ldots l_k \). First, let \( v = v_0 \ldots v_n \) be a word over \( E \) that contains \( h \) as sub-path. We show that it is accepted by \( \text{Obs}(E, w, w') \). At some point \( v \) must contain the start of \( h \). That means, there is an index \( m \geq 0 \) such that \( v_m \ldots v_{m+k} = h \). As the transition relation is total, there are states \( s_0, \ldots, s_m \) such that \( s_0 \xrightarrow{(v_0, v_1)} \ldots \xrightarrow{(v_{m-1}, v_m)} s_m \) is an execution of \( \text{Obs}(E, w, w') \). At this point \( s_m \) is either \( \text{Conflict} \), in which the observer stays forever and which is an accepting state or the observer is in state \( \text{init} \) or some \( s_i \). In this case the above execution can be continued by

\[
s_m \xrightarrow{(l_0, l_1)} s_1 \xrightarrow{(l_{k-1}, l_k)} \text{Conflict} = s_{m+k} \xrightarrow{(v_{m+k}, v_{m+k+1})} \ldots \xrightarrow{(v_{n-1}, v_n)} \text{Conflict}
\]

Therefore, \( v \) is accepted by \( \text{Obs}(E, w, w') \). Now we show that if a word is accepted by \( \text{Obs}(E, w, w') \), it contains \( h \). Let \( u = u_1, \ldots, u_n \) be such a word. In order for a word to be accepted, the observer must stop in \( \text{Conflict} \). To reach \( \text{Conflict} \) it must have subsequently run through \( s_1, \ldots, s_{k-1} \). Therefore, an execution that produces \( u \) must contain the following part:

\[
\ldots \xrightarrow{(l_0, l_1)} s_1 \xrightarrow{\ldots} s_{k-1} \xrightarrow{(l_{k-1}, l_k)} \text{Conflict} \rightarrow \ldots
\]

Therefore \( u \) must contain \( l_0, \ldots, l_k \) which means \( h \) is a sub-path of \( u \). \( \square \)
Lemma 5.6. If \( w \) is a word over the alphabet \( E \) and \( w' \) is an infeasible core of \( w \), then every word \( v \) over \( E \) that contains the hull \( h \) of \( w' \) in \( w \) as a sub-path, represents an infeasible path.

Proof. We know that \( h \) represents an infeasible path because \( w' \) is an infeasible core. Now Lemma 4.4 exactly states the proposition. \( \square \)

With those two lemmas established we can justify our construction of LCFG observer automata.

Proposition 5.2. Given an LCFG \( P \) with edge-set \( E \) and a spurious word \( w \) with infeasible core \( w' \) then \( \text{Obs}(E, w, w') \) separates \( w \) from \( P \).

Proof. We first show that \( \text{Obs}(E, w, w') \) accepts \( w \). Lemma 5.5 tells us that \( \text{Obs}(E, w, w') \) accepts exactly the words that contain the hull of \( w' \) in \( w \). Therefore it obviously accepts \( w \).

Now we show that \( \mathcal{L}(\text{Obs}(E, w, w')) \cap \mathcal{L}(P) = \emptyset \). With Lemma 5.6 we can argue that all the words that \( \text{Obs}(E, w, w') \) accepts, represent infeasible paths, because they contain the hull of an infeasible core. On the other hand, Proposition 5.1 ensures that \( \mathcal{L}(P) \) contains only feasible paths. Therefore, \( \mathcal{L}(\text{Obs}(E, w, w')) \cap \mathcal{L}(P) = \emptyset \). \( \square \)

We have shown how to construct observers given an infeasible core and have argued that this construction is “good” as we can produce a sequence of refined abstractions of our LCFG. So far, we have not detailed how to find those infeasible cores. This is what the next section is about.

5.5 Computing Infeasible Cores

5.5.1 Computing a Single Infeasible Core

Given a counter-example word \( w = l_0 \ldots l_n \), we want to determine if it represents a feasible path in the program and if not, calculate infeasible cores in order to construct LCFG observer automata. We first show how we can decide the feasibility of a path and calculate one infeasible core. After that we describe how to find potentially more infeasible cores in the same path.

Just deciding the infeasibility of a path \( p \) is pretty straight forward. All that needs to be done is to calculate the weakest precondition with regard to \( true \) and check if \( wp(p, true) \) is unsatisfiable. This check can be performed by using an SMT solver. If it is unsatisfiable then \( p \) is an infeasible path. Yet, we want not only decide infeasibility but compute an infeasible core. We will do this by computing unsatisfiable subformulas of the weakest precondition in CoNF. The correctness of this course of action will be proved at the end of this section. However, we need a lemma before we can give a proof of the claim.
To simplify the notation for the next steps, we define the shortcuts $F_C^S$ and $F_S$, where $C$ and $S$ are sets of integer indices. The idea of these shortcuts is that they filter a formula in CoNF. The shortcuts are defined as follows:

$$F_C^S(\bigwedge_{i=1}^{n+1} \varphi_i \sigma_{i-1} \ldots \sigma_0) = \bigwedge_{i \in C \cap \{1, \ldots, n+1\}} \varphi_i F_S(\sigma_{i-1} \ldots \sigma_0)$$

$$F_S(\sigma_1 \ldots \sigma_{k-1} \sigma_k) = \begin{cases} F_S(\sigma_1 \ldots \sigma_{k-1}) \sigma_k & \text{if } k \in S \\ F_S(\sigma_1 \ldots \sigma_{k-1}) & \text{otherwise} \end{cases}$$

where $\epsilon$ denotes the empty sequence of substitutions.

**Lemma 5.7.** Given a path $p = s_1; \ldots; s_n$ and a formula $\varphi_{cf}$ that is in CoNF for $p$. If for $S = \{0, \ldots, n\}$ and some $C = \{i_1, \ldots, i_k\}$ the subformula $F_S^C(\varphi_{cf})$ of $\varphi_{cf}$ is unsatisfiable, then the weakest precondition $wp(p', true)$ of the path projection $p' = s_{i_1'; \ldots; s_{i_k'}}$ is also unsatisfiable.

**Proof.** We show the contraposition of the claim, i.e. if $wp(p', true)$ is satisfiable, then so is $F_S^C(\varphi_{cf})$. Thus, assume $wp(p', true)$ is satisfiable. Let $S' = C = \{i_1, \ldots, i_k\}$. Then

$$F_C^S(\varphi_{cf}) = \bigwedge_{j=1}^{k} \varphi_{ij} \sigma_{i_{j-1}} \ldots \sigma_{i_1}$$

is equivalent to $wp(p', true)$. Obviously, one can transform $F_C^S(\varphi_{cf})$ to $F_S^C(\varphi_{cf})$ by iteratively adding indices to $S'$. We show that this operation maintains satisfiability.

Assume for some set $S''$ with $S' \subseteq S'' \subseteq S$ that $F_{S''}^C(\varphi_{cf})$ is satisfiable and let $l \in S \setminus S''$. We have to prove that $F_{S'' \cup \{l\}}^C(\varphi_{cf})$ is also satisfiable. We now partition the sets $C$ and $S''$ along $l$.

$$C^c = \{c \in C \mid c < l\}$$
$$C^+ = \{c \in C \mid c > l\}$$
$$S^c = \{s \in S'' \mid s < l\}$$
$$S^+ = \{s \in S'' \mid s > l\}$$

Using this partitioning we have the equation

$$F_{S''}^C(\varphi_{cf}) = F_{S''}^{C^c}(\varphi_{cf}) \land F_{S''}^{C^+}(\varphi_{cf})$$

$F_{S''}^{C^+}(\varphi_{cf})$ in turn can be rewritten as

$$\bigwedge_{i \in C^+} \varphi_i F_S(\sigma_{i-1} \ldots \sigma_0) F_S(\sigma_{i-1} \ldots \sigma_0)$$
Proposition 5.3. Let \( \varphi_{cf} \) give us a procedure to compute infeasible cores. As we can use the SMT solver to compute unsatisfiable subformulas, this which is in CoNF for a path, we also have an infeasible core of this path. Proposition 5.3 guarantees that by computing an unsatisfiable subformula of a formula

\[
F^C_{S''}(\varphi_{cf}) = F^C_{S''}(\varphi_{cf}) \land \bigwedge_{i \in C'} \varphi_i F^C_{S'}(\sigma_{i-1} \ldots \sigma_0) F^C_{S'}(\sigma_{i-1} \ldots \sigma_0)
\]

We can now express \( F^C_{S'' \cup \{l\}}(\varphi_{cf}) \) by

\[
F^C_{S'' \cup \{l\}}(\varphi_{cf}) = F^C_{S''}(\varphi_{cf}) \land \bigwedge_{i \in C'} \varphi_i F^C_{S'}(\sigma_{i-1} \ldots \sigma_0) \sigma_l F^C_{S'}(\sigma_{i-1} \ldots \sigma_0)
\]

With Lemma 2.4 we can now argue that \( F^C_{S'' \cup \{l\}}(\varphi_{cf}) \) is satisfiable if \( F^C_{S''}(\varphi_{cf}) \) is, supposing we can show the necessary properties for \( \sigma_l \).

\( \sigma_l \) is of the form

\[
\{\{m_i \mapsto \text{update}(m_i, lv_i, \theta_{l_i})\}_i\}
\]

where \( \theta_{l_i} \) are variables that only appear again in \( \varphi_1 \). We define the left-inverse

\[
\sigma_l^{-1} = \{(\theta_{l_i} \mapsto \text{access}(m_i, lv_i))_i\}
\]

with \( \text{dom}(\sigma_l^{-1}) = \{(\theta_{l_i})_i\} \). This is a left-inverse because by the axioms of the memory model one can deduce that

\[
\text{update}(m_i, lv_i, \text{access}(m_i, lv_i)) = m_i
\]

As \( l \) is not in \( S'' \) it is also not in \( S' \) and therefore not in \( C \). This however means that \( \varphi_1 \) is not a part of \( F^C_{S''}(\varphi_{cf}) \). Therefore \( \sigma_l^{-1} \) fulfills the condition that \( \text{dom}(\sigma_l^{-1}) \cap \text{vars}(F^C_{S''}(\varphi_{cf})) = \emptyset \). As all \( \theta_{l_i} \) are fresh, they are also never part of the domain or the range of another substitution \( \sigma_l \). Thus the two remaining preconditions for Lemma 2.4 are fulfilled.

Now we can formulate an important result of this section. Proposition 5.3 guarantees that by computing an unsatisfiable subformula of a formula which is in CoNF for a path, we also have an infeasible core of this path. As we can use the SMT solver to compute unsatisfiable subformulas, this finally gives us a procedure to compute infeasible cores.

**Proposition 5.3.** Let \( p = s_1; \ldots ; s_n \) be a path and \( \varphi_{cf} \) a formula in CoNF for \( p \). If for \( C = \{i_0, \ldots , i_k\} \) and \( S = \{1, \ldots , n\} \) the formula \( F^C_S(\varphi_{cf}) \) is an unsatisfiable subformula of \( \varphi_{cf} \), then \( p' = p_{i_0}; \ldots ; p_{i_k} \) is an infeasible core of \( p \).

**Proof.** \( p' \) is obviously a projection of \( p \) and the fact that \( p' \) is infeasible follows directly from Lemma 5.3. We have to show that for some projection \( p'' = p_j; \ldots ; p_m \) of \( p \) such that \( p'' \) is a projection of \( p'' \), \( p'' \) is infeasible. Because of the projection relationships we have that \( C \subseteq C' = \{j_0, \ldots , j_m\} \) and therefore \( F^C_S(\varphi_{cf}) \) is a subformula of \( F^C_S(\varphi_{cf}) \). As \( F^C_S(\varphi_{cf}) \) is unsatisfiable, \( F^C_S(\varphi_{cf}) \) is unsatisfiable as well and by Lemma 5.7, \( p'' \) is infeasible. \( \square \)
5.5 Computing Infeasible Cores

5.5.2 Computing Many Infeasible Cores

An infeasible path may be infeasible due to many reasons. Therefore, there might be many (independent) unsatisfiable subformulas of the weakest precondition w.r.t. true. Furthermore, we are interested in finding minimal unsatisfiable subformulas in order to minimize the number and the size of the observers. SMT solvers usually can be instructed to deliver an unsatisfiable subformula but, as mentioned before, they usually do not guarantee minimality. On the other hand, finding all minimal unsatisfiable subformulas requires exponentially many calls to the SMT solver in the worst case. (For algorithms see for example [20] and [38]). To avoid this, we employ a simple algorithm that has only linear complexity but does not find all minimal unsatisfiable subformulas. The algorithm exploits the ability of state-of-the-art SMT solvers to allow asserting and retracting constraints incrementally. The main idea of the algorithm is to use the solver’s capabilities to find unsatisfiable subformulas and minimize them using the incremental algorithm described in [20]. As the unsatisfiable subformulas suggested by the solver are usually small, it is practically feasible to employ the exponentially complex algorithm on them. We decided to use this algorithm as it makes use of the incremental solving capability of modern SMT solvers. Algorithm 1 shows the pseudo-code for the high-level algorithm and Algorithm 2 shows the pseudo-code for the minimization algorithm. We will now give a more detailed description of the algorithms.

Description of the algorithms

We assume we have an interface to the SMT solver consisting of CreateContext to generate a new logical context, Assert to assert a constraint into a logical context, Sat to check the satisfiability of the current context, Push and Pop, to push respectively pop logical contexts to or from the context stack and UnsatSub to extract a (small) unsatisfiable subformula from the current context, if the latter is unsatisfiable. In the following, we will represent formulas as sets of constraint and apply the set-operators on them.

FindUnsatSub is the high level algorithm to find minimal unsatisfiable subformulas. The idea of the algorithm is to assert the conjuncts of the weakest precondition (in CoNF) into the logical context of the SMT solver one after another, until there are no conjuncts left or the context becomes unsatisfiable. If the context is unsatisfiable, the solver returns an unsatisfiable subformula. This subformula, however, is not guaranteed to be minimal. In fact, it may even contain several minimal unsatisfiable subformulas. The algorithm now calls MinUnsatSubs to find all minimal unsatisfiable subformulas that are part of the subformula the solver returned. In order to find more minimal unsatisfiable subformulas, we retract one constraint from each of the minimal unsatisfiable subformulas we found. The context is now
satisfiable again and we can go on adding new constraints. The function \texttt{Eliminate} encapsulates the heuristic that retracts the constraints.

**Algorithm 1** The high level algorithm to find unsatisfiable cores

```plaintext
function FindUnsatSub(\varphi_1, \ldots, \varphi_n)
    Ctx ← CreateContext  \triangleright current logical context
    C ← \emptyset  \triangleright result set of unsat cores
    Next ← 1
    while Next ≤ n do
        Assert(Ctx, \varphi_{\text{Next}})
        if \neg \text{Sat}(Ctx) then
            S ← UnsatSub(Ctx)
            C' ← MinUnsatSub(S)
            C ← C \cup C'
            Ctx ← Eliminate(Ctx, C')
        end if
        Next ← Next + 1
    end while
    return C
end function
```

\texttt{MinUnsatSubIter} is based on an algorithm by de la Banda et al. [20]. The difference is that it uses just one logical context and makes use of the stack based solver interface. We basically use the same notation, but instead of two logical contexts we use just one: \texttt{Ctx}. As the proof of correctness is rather lengthy we won't fully reproduce it here. See [20] for the proof. We will, however, explain the intuition behind the algorithm. A call to \texttt{MinUnsatSubIter}(Ctx, D, P, T, C, \texttt{St.Size}) finds all minimal unsatisfiable subsets of the constraints in \texttt{P} that contain \texttt{D} as a subset. The precondition is that the context \texttt{Ctx} contains exactly the constraints in \texttt{T} and is currently satisfiable. Furthermore, by a \texttt{St.Size}-fold \texttt{Pop} on \texttt{Ctx} one can retrieve a context that exactly mirrors \texttt{D}. \texttt{C} is the set of minimal unsatisfiable subformulas found so far. The postcondition is that the topmost \texttt{St.Size} contexts have indeed been popped and thus \texttt{Ctx} mirrors \texttt{D}. In the algorithm we assert the constraints from \texttt{P} one by one and check the satisfiability. If the context never turns unsatisfiable we establish the postcondition by popping the nessecary amount of contexts and are done. Otherwise we retract the last constraint by popping the last context from the stack (which was satisfiable) and recursively call the function, now excluding the last constraint. This means, we look for unsatisfiable subformulas that do not contain \varphi. In the rest of the function we look for unsatisfiable subformulas that contain \texttt{D} \cup \varphi. After the recursive call, we know that \texttt{Ctx} now represents \texttt{D}. We first check if \texttt{D} \cup \varphi is satisfiable. If so, we recursively search subformulas that contain
$D \cup \varphi$. Otherwise we have to check if we already found an unsatisfiable subformula that is a subset of the one we found now. If this is the case, we can discard the current one, because it is not minimal. Otherwise we add it to the set of found minimal unsatisfiable subformulas $C$.

**Some hints on the implementation**

In the implementation, we employ some optimizations in order to speed up the process of finding the unsatisfiable subformulas. Most optimizations make use of the capabilities of the solver for incremental solving. One of those optimizations is to assert constraints of the form $p_i \iff \varphi_i$ for each conjunct of the weakest precondition formula. This introduces proxies $p_i$ for all conjuncts. After that the context is checked for satisfiability and the pushed on the stack. From now on only the proxies are asserted and when checking the context, the solver may make use of the information it gathered during the first checking run. As the solver may re-use a lot of information, the subsequent checking runs are much faster. Another optimization is to only run a satisfiability check on the current context if we know it can be unsatisfiable. For example, if there is no proxy asserted that represents an `assume` statement, then we do not have to run the solver.
5.5 Computing Infeasible Cores

Algorithm 2 The algorithm to find all minimal unsatisfiable cores

function MinUnsatSub(P)
    return MinUnsatSubIter(CreateContext, ∅, P, ∅, ∅, 0)
end function

function MinUnsatSubIter(Ctx, D, P, T, C, St_Size)
    Ctx_Sat ← True
    ▷ Assert until context unsat or no more constraints left

    while result ∧ ∃ ϕ ∈ P \ T do
        T ← T \ {ϕ}
        Push(Ctx)
        St_Size ← St_Size + 1
        Assert(Ctx, ϕ)
        Ctx_Sat ← Sat(Ctx)
    end while

    if Ctx_Sat then
        ▷ establish postcondition
        return C
    end if

    Pop(Ctx, 1) ▷ restore the last sat context

    C ← MinUnsatCores(Ctx, D, P \ {ϕ}, T \ {ϕ}, C, St_Size - 1)
    Push(Ctx)
    D ← D ∪ ϕ
    Assert(Ctx, ϕ)

    if ¬Sat(Ctx) then
        ▷ restore the last sat context
        return C
    end if

    if C ← MinUnsatCores(Ctx, D, P \ {ϕ}, D, C, 0)
    Pop(Ctx, 1)
    return C
end function
Chapter 6

Experimental Results

We implemented a prototype of the SMT-based false-positive elimination procedure (SFPE) as an optional addition to the Goanna Static Analysis Tool and evaluated it by analyzing two projects from the SATE 2010 tool exposition [1].

6.1 Implementation

The procedure has been implemented in the functional programming language O’Caml. For our experiments we used the state-of-the-art SMT solver Z3 [21], which is free for use for academical purposes. There are other solvers available (e.g. [12],[23]) that we could have used as well. The main reason why we chose Z3, is its reported performance, especially on arrays and linear arithmetic (see [2]). Furthermore, Z3 comes with some additional tools (e.g. the Axiom Profiler which makes optimization and debugging easier) and has an O’Caml programming interface. To enable switching between different solvers (which we didn’t experiment with, however) we introduced an abstraction layer between the solver and the rest of the implementation.

Goanna currently uses two alternatives for model checking: First the open source symbolic model checker NuSMV and second an in-house development that is optimized for model checking tasks that occur in static analysis. For technical reasons we could only use the NuSMV variant for the refinement loop. The evaluation shows that most of the time of the SFPE procedure is spent by the model checker. Therefore, a switch to the higher performance model checker could boost the performance of the whole SFPE procedure.

A second issue in the implementation is that we didn’t connect the FPE procedure to the interprocedural framework of Goanna. This means we do not use the function summaries for the semantics yet. Instead, we use a generic function summary. As this is too pessimistic in some cases this results in some misses of false-positives.
6.2 Evaluation

We report on the findings for Wireshark 1.2.9 and Dovecot 2.0 beta6. They seem suited for our purposes because they contain both: optimized, rather low level C-code as well as high level code for user interfaces. Furthermore, both programs have rather large code bases of several hundred thousands source lines of code, which seems to be a size that compares to projects in the industrial context. We experimented with other open source programs of different size with very similar results as for the two mentioned projects. We did not, however, run tests with industrial projects or source code for embedded devices or drivers.

6.2.1 Preciseness

During the first experiments we found several cases where our procedure was too imprecise. As it assumed no prior knowledge about global variables and function parameters, it admitted aliasing between global pointers or pointer parameters and local variables at the beginning of a function. Although this situation is possible in principle, we decided to exclude this scenario by adding an additional background axiom that asserts that local variables are not pointed to when they are created.

Goanna performs several checks on a program. However, in the default set of checked properties, we only found false-positives for two types of properties. This is explained by the fact that not all checks may yield false-positives that can be refuted by analysis of the counter-example. For example, there is a check in Goanna to find out if some location is reachable for which a prior analysis determined that a comparison always yields true. Normally, the location is reachable in some way unless it is dead code. We will therefore restrict the analysis of the precision to those two properties. This seems legitimate as those two properties have shown to be the main causes of false-positives. Furthermore, both have rather long witnesses which means it may be extremely difficult to check for false-positives manually. The first of these checks is Null-Pointer dereferencing. It checks if some variable is possibly assigned `Null` and is later dereferenced. The second check searches for uses of variables where the variable is possibly uninitialized at that point. Table 6.1 summarizes the results. For Wireshark, Goanna produces 98 warnings of the two types mentioned above. 61 warnings are about uninitialized variables and 37 warnings about possible Null-Pointer dereferencing. We are able to identify 48 warnings as false-positives. This means around 49% of the produced warnings can be eliminated. We manually checked if any of the removed warnings were wrongly removed. As far as we can tell, this was not the case. The results for Dovecot were very similar to the ones for Wireshark. The percentage of false-positives was slightly higher (50.6%). In both experiments, there were several cases
6.2 Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Wireshark 1.2.9</th>
<th>Dovecot 2.0 beta6</th>
</tr>
</thead>
<tbody>
<tr>
<td>issued warnings</td>
<td>98 (61+37)</td>
<td>75 (36+39)</td>
</tr>
<tr>
<td>fp’s removed</td>
<td>48 (27+21)</td>
<td>38 (21+17)</td>
</tr>
<tr>
<td>% fp’s of original warnings</td>
<td>49.0 %</td>
<td>50.6 %</td>
</tr>
<tr>
<td>correctly identified as fp</td>
<td>48 (100 %)</td>
<td>38 (100 %)</td>
</tr>
</tbody>
</table>

Table 6.1: Results of running the SFPE procedure on Wireshark and Dovecot

where false-positives have still been missed. The main reasons for this are the following: Due to the incompleteness of the procedure e.g. with regard to bit-level functions, a path could not be identified as infeasible. A second reason is that the loop limit was reached before a warning could be refuted. Moreover, due to the missing interprocedural information, the effects of a function are sometimes approximated in a too pessimistic way. A last reason is that an abortion condition could not be identified as such as it was wrapped in some function.

6.2.2 Performance

We used Goanna without interprocedural analysis with the default properties enabled. The loop limit was set to 20. This means if after 20 iteration of the refinement loop a warning could not be refuted, it is considered a valid warning. This happened 11 times in Wireshark but only once in Dovecot. Typically this happened in very long functions with a lot of conditions. The timeout in Goanna was set to 120 seconds. This means, if the analysis of a file takes longer than that, it is terminated. For the timing information we only used the cases where the analysis terminated before the timeout. This may introduce a bias as a timeout may be triggered due to an excessive SFPE. But the timeout did not happen very often. In Wireshark, it happened in 12 cases (which accounts for roughly 1%), in Dovecot only in one case. However, in Dovecot, the SMT solver timeout was triggered in some cases. This happens when the satisfiability of a formula could not be determined in a short time (~2s). Table 6.2 summarizes the performance results. The overall running time for Wireshark was around two and a half hour, for Dovecot (as it is much smaller) around 17 minutes. Around 15% of that time was spent in the false-positive elimination procedure in Wireshark (30% in Dovecot). This value depends on the number of properties that are enabled in Goanna and of course on the source code. The more warnings are issued, the more time is spent to check for false-positives. In other experiments we have measured a maximum factor of 50%. This means the complete analysis took about twice the time with the SFPE procedure enabled than without. But even this seems to be an acceptable trade-off.
### 6.2 Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Wireshark 1.2.9</th>
<th>Dovecot 2.0 beta6</th>
</tr>
</thead>
<tbody>
<tr>
<td>total running time (no timeout)</td>
<td>8815s (2h 27min)</td>
<td>1025s (17 min)</td>
</tr>
<tr>
<td>time spent in SFPE</td>
<td>1332s (15%)</td>
<td>302s (29.5 %)</td>
</tr>
<tr>
<td>% of SFPE time in dec. proc.</td>
<td>10.5 %</td>
<td>12.2 %</td>
</tr>
<tr>
<td>% of SFPE time in mod. check.</td>
<td>87.5 %</td>
<td>86.3 %</td>
</tr>
<tr>
<td>number of timeouts</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>number of loop exceedings (20)</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>number of solver timeouts</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6.2: Performance of the false-positive elimination procedure for Wireshark and Dovecot

We noticed, however, that the SFPE spends most of the time with model checking and not with the decision procedure.

#### 6.2.3 Comparison with the Interval Based FPE

So far, we did not run comparative tests with the interval based false-positive elimination procedure. The reasons for this are mainly technical. As the interval based approach was not maintained any more in Goanna we could not run it with the current Goanna version.
Chapter 7

Related Work

Our false-positive elimination procedure is based on the principles brought forward in the CEGAR approach. The CEGAR approach is developed in the classical paper [17] by Clark et al. There are many results on applying CEGAR to software model checking. Most of them are based on predicate abstraction. The idea is to abstract a program as a boolean program with regard to a set of predicates. In the case of a spurious counter-example new predicates are added and a new boolean program is calculated. Successful applications of predicate abstraction include Blast [30] and SLAM ([10],[9]. SLAM is developed by Microsoft for the static analysis of device drivers. The refinement component in SLAM, called Newton [11], performs a symbolic execution of the path similar as we do although they compute strongest postconditions instead of weakest preconditions. However, predicate abstraction does not fit well in the purely syntax-based approach that Goanna pursues as it performs abstraction on data. The current false-positive elimination procedure in Goanna is described in [27]. It comes with the problems outlined in Chapter 3.

There are several suggestions for a memory model for the C programming language. In the context of verification, [43] and [44] develop a memory model based on separation logic. [37] uses a C memory model to verify program transformations. [18] presents a memory model for C used in conjunction with a verifying compiler. Although the last one is designed to work together with SMT solvers, those memory models are generally not abstract enough to be used in our context of static analysis. There is a line of work to devise specialized logics for dealing with pointers and encode them in other logics like first-order logic. An example for this is [36]. Those efforts are tailored for reasoning about more complex properties of programs (like properties of linked lists) where a more expressive logic is of advantage. As we do not need to reason about such properties we wanted to avoid the overhead of a specialized logic. Another approach is to build a memory model not for the actual source language but for some low-level
language. [42] develop a memory model used for the low level language of the LLVM compiler infrastructure. Again this does not match the syntax driven approach of Goanna.

The component-as-array-model is used in [32] and [28]. Both have significantly influenced the development of our memory model and the l-value semantics. The component-as-array-model is furthermore used in [39] in the context of interactive theorem proving with Isabelle/HOL.

Using predicate transformers and especially weakest preconditions in static analysis is very common. [29] compute weakest precondition in the ESC/Java tool. ESC/Java is an extended static checker that works by defining annotations like pre- and postconditions that a programmer can attach to some piece of code. Therefore it is not a fully automatic tool like Goanna. Snugglebug [16] is a static analysis tool that combines weakest precondition computation with a component-as-array-based memory model, although the memory model is not described in detail in the paper. However, Snugglebug does not do model checking. As pointed out above, SLAM uses strongest postconditions as vehicle to analyse counter-examples. However, Ball shows in [6] how a framework as used in SLAM may be formulated using weakest preconditions together with Craig interpolants. Again this is based on the general idea of predicate abstraction. In SLAM2 [8], preconditions are used to find several explanations for the infeasibility of a path in order to speed up the whole analysis. For the same reason we compute several infeasible cores. However in SLAM2 the advantage seems to stem from the combination of backwards and forwards analysis and not from the use of unsatisfiable subformulas.

There are many examples of using SMT solvers in the realm of software model checking. [3] and [19] use an SMT solver as reasoning engine for bounded model checking. In order to reduce the number of paths that have to be explored in BMC, Yang et al. [45] propose a method, called Dynamic Path Reduction, to prune infeasible paths using SMT solvers. This is very similar to our approach, as Yang et al. also try to find infeasible sub-paths by finding unsatisfiable subformulas in the weakest precondition of a path. However, Yang et al. describe their method only for a simple language without pointers and therefore may employ the standard definition of wp.

To our knowledge there are no works yet that use weakest preconditions and SMT solver to compute explanations for a refinement procedure in conjunction with syntax-based software model checking for a real world programming language like C including language features like pointers.
Chapter 8

Conclusion

8.1 Summary

In this thesis we describe a procedure to eliminate false-positives that arise in a model-checking based static analysis as performed by Goanna. We introduce the concept of paths through a C program and develop a language to represent such paths. We give a semantics of that language using a memory model in order to capture language features like pointers. The semantics transforms a path into a first-order formula using the weakest precondition predicate transformer. Based on the path language we develop a method for a counterexample-guided-path-reduction. We therefore represent a program as a labelled control flow graph. An LCFG is assigned a meaning by an automaton. We give a finite abstraction of this automaton and iteratively refine this abstraction using observer automata. We prove the correctness of this approach in the sense that we show that each refinement fulfills the invariant that it is an abstraction of the original model but allows a reduced set of possible executions. For the computation of observer automata we use infeasible cores of counter-example paths and therefore present a way to use an SMT solver to compute these cores.

We implemented the technique in the framework of Goanna and used it to analyse two large open-source projects. The evaluation shows that for properties that are dependent on paths, like uninitialized variables, the SMT based false-positive analysis significantly reduces the number of false-positives and therefore contributes to the usability of Goanna.

8.2 Further Work

There are several issues where further investigations seem promising. Concerning the semantics it would be interesting to evaluate using a bit precise semantics. This could be achieved using the theory of bitvectors for which many SMT solvers provide decision procedures. This would eliminate one
8.2 Further Work

reason for the incompleteness of the semantics. Also, a more precise type
system would enable us to model type casts more adequately.

As the evaluation shows, the bottleneck of the current SFPE procedure
is not the decision procedure but the model checking. Therefore, a more
precise decision procedure seems affordable even for the price of a slight
loss of performance. Performance could be increased however by performing
some kind of path slicing or cone-of-influence computation on a path before
handing it over to the decision procedure. The idea is that certain fragments
of a path that cannot contribute to its infeasibility are removed from it. As
the weakest precondition would be smaller, this would speed up the process
of deciding its satisfiability.

A problem that we did not mention so far is the handling of loops. In
some cases, the SFPE cycle degrades to an iterative unrolling of a loop. As
the loop might be long or even infinite, the loop limit is reached quickly with-
out much information gained. Therefore, research in this direction would be
of great value.

Another line of improvement would be to not only check a path for its
feasibility but also check if a given path really violates a given property. E.g.
if some variable really may be Null at a given program point. This would
make it possible to use the SFPE procedure for a wider range of properties.
An important question here would be how to integrate this with the path
reduction approach.

As Goanna gathers a lot of information by using traditional classic anal-
ysis techniques before running the actual model checking, it would be inter-
esting to see if one could use this information (e.g. the possible values of
variables or points-to-sets) to increase the precision of the decision proce-
dure.

As the memory model heavily influences the shape of the generated for-
mulas there is probably also room for improvement. For example, the fact
that we use an algebraic datatype as array index seems to slow down the
decision procedure. We therefore evaluated a flat memory model that used
integers as array indices. However, we ran into problems as it was not pre-
cise enough. [7] propose a set of axioms for handling pointer predicates. It
would be interesting if one could use a similar set of axioms in our case, too.
Bibliography


