Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

Compiler Optimizations using Symbolic Abstraction

by
Fabian Ritter

submitted
November 30, 2015

Reviewers:

1. Prof. Dr. Sebastian Hack
2. Prof. Dr. Jan Reineke

Advisor:

Tomasz Dudziak M.Sc.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

With increasing program complexity, a vital requirement for compilers is to produce efficient code with as few instructions as possible. This requires program analyses that are expressive enough to prove relevant program properties, but still effectively computable.

In the last decades, a large variety of such program analyses has been developed. The better availability of computational power tempts analysis designers to create more expensive and powerful analyses, often combined from several simpler ones. For most such combinations, it is possible to construct examples where they outperform any sequential execution of the contributing analyses.

However, it has not yet been investigated whether these examples are plausible cases that are relevant in practice or just artificial constructs that do not appear in production code and therefore whether combining the analyses is profitable in general.

In this thesis, Sprattus, a framework for program analysis using symbolic abstraction, is used and extended to implement several program analyses for compiler optimization. It reduces program analysis to the Satisfiability Modulo Theories problem and utilizes a state-of-the-art theorem prover to derive non-trivial program properties. Sprattus provides the necessary tools for easily creating analyses of arbitrary complexity and expressiveness. The efficiency of combinations of the developed analyses is evaluated on a set of relevant benchmarks.

Acknowledgments

To begin with, I want to thank Professor Sebastian Hack for supervising this thesis and Professor Jan Reineke for reviewing it.

Furthermore, I am very grateful to my advisor Tomasz Dudziak. I want to thank him for all the time that he spent answering my questions, for reading and criticizing my thesis and for all the insightful discussions.

I also want to express my gratitude to the members and the current students of the Compiler Design Lab for their helpful suggestions during my work on this thesis and for providing me a pleasant and productive working environment.

Moreover, I thank my proofreaders Immanuel Haffner, Sebastian Hahn, Franziska Müller and Matthias Ritter for their useful feedback.

Finally, I want to thank everyone else who supported me in my work on this project.

Contents

List of Figures

1 Introduction

With increasing program complexity, it becomes more and more vital for compilers to produce efficient code. Unnecessarily executed instructions are an obvious source for performance loss and should thus be avoided.

Unfortunately, determining for an instruction whether it is unnecessary is in general undecidable. Hence, a significant part of designing compiler optimizations is creating program analyses with an acceptable compromise between expressiveness and feasibility.

As the hardware evolves, more and more computational power becomes available. Therefore, more complex program analyses have been developed over time, permitting powerful program transformations. It is tempting to construct such analyses by combining simpler analyses. For most such combinations, it is possible to find artificial examples where they are able to produce more precise results than the sequential execution of the contributing analyses.

For such an example, consider the program from [Click and Cooper, 1995] that is depicted in Figure 1.1. It works on three variables `x`, `y` and `z` in a loop with a statically unknown condition with no side effects.

At the beginning, `x` is initialized to 1 whereas `y` and `z` have the same, at compile time unknown value. By studying the conditional statements in the loop, one can observe that the value of `x` is modified if the value of `y` or `z` has been modified before. Conversely, the value of `y` is left unchanged unless `x` is modified before.

As the statement in line 9 does not modify the value of `x` if it is 1 before, the code inside the loop does not modify the variables at all and could thus be removed by an optimizing compiler.

```
1  int foo(int z)
2  {
3    int x = 1;
4    int y = z;
5    while (dontknow())
6    {
7      if (y != z)
8        x = 2;
9      x = 2 - x;
10     if (x != 1)
11       y = 2;
12   }
13   return x;
14 }
```

Figure 1.1: Example program, from [Click and Cooper, 1995].

However, automatically discovering this fact is a non-trivial task: typically, common compiler optimizations are able to determine variables with statically constant values on the one hand or invariant relations between variables on the other hand. For this example, neither of these abilities alone is sufficient to permit removing the loop. Whereas the former of the mentioned program analyses is unable to prove the statement in line 8 unreachable, the latter fails to mark the statement in line 11 as unreachable, both ultimately providing no additional information.

The approach described in [Click and Cooper, 1995] introduces a new program transformation that combines both aforementioned optimizations and is able to prove `x` constant and the loop to be dead in this example.

However, it has not yet been investigated whether these examples are plausible cases that are relevant in practice or just artificial constructs that do not appear in production code.

This thesis uses and extends Sprattus, a program analysis framework based on concepts from abstract interpretation [Cousot and Cousot, 1977] and symbolic abstract interpretation [Thakur, 2014] to implement expressive analyses for aggressive code transformations.

The modular construction of Sprattus provides a powerful interface for the design of complex, combined analyses. This permits the rapid prototyping of analyses and compiler optimizations as it is described for several examples in this thesis.

The performance of the so constructed code transformations will answer questions about the efficiency of the Sprattus framework and the underlying algorithms as well as about the use of complex, combined analyses in optimizing compilers.

The overall structure of this thesis can be summarized as follows: Chapter 2 gives an overview of the relevant theoretical concepts such as (symbolic) abstract interpretation and satisfiability modulo theories. In chapter 3, the external frameworks that found applications in the thesis are presented.

The main contributions of this thesis are described alongside the general Sprattus framework in which they are integrated in chapter 4. An evaluation of the approach as described in the preceding chapters is performed in chapter 5 on a set of relevant benchmarks.

Chapter 6 sets the work that is presented in this thesis into context with previous work on related topics. Finally, chapter 7 draws conclusions from the contributions of the thesis.

2 Theoretical Background

This thesis uses and evaluates concepts from Abstract Interpretation, Satisfiability Modulo Theories and Symbolic Abstract Interpretation that are described in this chapter.

2.1 Abstract Interpretation

Abstract Interpretation is a formal framework for program analysis introduced in [Cousot and Cousot, 1977]. It formalizes the idea of systematically discarding information about a subset of the observable program properties in favor of non-trivial analysis results despite the general undecidability of program analysis.

This section includes several basic definitions and results for Abstract Interpretation. For more detailed background information and proofs consider [Nielson et al., 1999, Chapter 4].

2.1.1 Preliminaries

The necessary abstraction of the program semantics is formalized with two opposing domains: on one hand the *concrete domain* that closely models the information of concrete program runs or program states. On the other hand, there is the *abstract domain*, specifically tailored to capture the information from the concrete domain that is relevant for the desired analysis.

The concrete and abstract domains are required to have certain algebraic structures specified in the following definitions.

Definition 2.1. A partially ordered set $(L, \sqsubseteq)$ is said to be a *complete lattice* if every subset P of L has a unique *greatest lower bound* (or *meet*) $\bigsqcap P$ and a unique *least upper bound* (or *join*) $\bigsqcup P$ in L.

Thus, a complete lattice contains two (not necessarily distinct) elements $\bigsqcap L = \bigsqcup \varnothing = \bot$ and $\bigsqcap \varnothing = \bigsqcup L = \top$.

For arguing about termination and runtime of many algorithms operating on complete lattices, a measure for the complexity of the lattice is necessary:

Definition 2.2. A subset $Y \subseteq L$ of a complete lattice $(L, \sqsubseteq)$ is said to be a *chain* of $(L, \sqsubseteq)$ if

$$\forall l_1, l_2 \in Y.\ (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1).$$

An *ascending chain* of a complete lattice $(L, \sqsubseteq)$ is a sequence $(a_i)_{i \in \mathbb{N}}$ of elements of L such that $\forall i \in \mathbb{N}.\, a_i \sqsubseteq a_{i+1}$ holds.

The *height* of a complete lattice $(L, \sqsubseteq)$ is the maximal number of distinct elements in any chain of $(L, \subseteq)$ if there is no infinite chain in $(L, \sqsubseteq)$, and ∞ otherwise.

$(L, \sqsubseteq)$ is said to satisfy the *ascending chain condition* if there exists no infinite ascending chain of L.

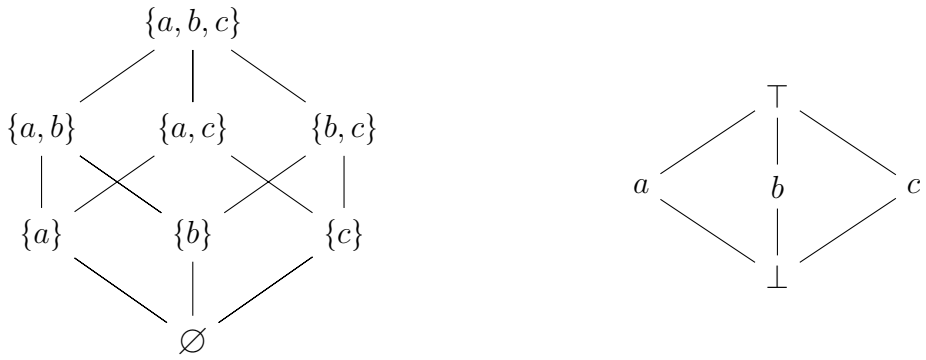
For applications in program analysis, it is convenient to generically construct complete lattices from arbitrary sets. In this thesis, the following two lattice constructions will be of particular relevance:

- The *power-set lattice* $(\mathcal{P}(S), \subseteq)$ of the set S is the set $\mathcal{P}(S)$ of all subsets of S ordered by set inclusion $\subseteq$. Meet and join coincide with set intersection $\cap$ and set union $\cup$, respectively. Hence, the set S itself is the $\top$ element of the lattice and the empty set $\varnothing$ is the $\bot$ element (cf. Figure 2.1a). Its height is $|S| + 1$ if the cardinality $|S|$ of the set S is finite and ∞ otherwise.

- To obtain the *flat lattice* $(S \cup \{\top, \bot\}, \sqsubseteq_{fl})$ of the set S, the set is extended with two explicit $\top$ and $\bot$ elements. The resulting set is partially ordered by the relation $\sqsubseteq_{fl}$ such that

  $$\forall s \in S.\ \bot \sqsubseteq_{fl} s \sqsubseteq_{fl} \top$$

  holds whereas the elements of S are not comparable according to $\sqsubseteq_{fl}$ (cf. Figure 2.1b). A flat lattice always has height 3.



(a) Power-set lattice (b) Flat lattice

Figure 2.1: Hasse diagrams of generic lattice constructions for the set $S = \{a, b, c\}$. The respective ordering relation is depicted by edges in the diagrams. It is the reflexive transitive closure of the edge relation, which is implicitly directed from higher to lower nodes.

For the set Ξ of possible concrete states of execution, the concrete domain $\mathcal{C}$ is defined as the power-set lattice $(\mathcal{P}(\Xi), \subseteq)$ of Ξ. The abstract domain $\mathcal{A}$ can be any complete lattice that can express the desired program properties. The ordering relations of the considered lattices are interpreted as an ordering in terms of the quality of information that is represented by the compared elements. $\top$ describes the absence of any restricting

information about the program, i.e. the set of all program states. In contrast to that, $\bot$ describes no program state at all. In an analysis that assigns lattice elements to program locations, a location that is mapped to $\bot$ exhibits no program behavior in any execution, i.e. it is unreachable.

The interpretation of $\mathcal{A}$ in terms of $\mathcal{C}$ is established by relating the domains with two functions as shown in Figure 2.2: An *abstraction function* $\alpha : \mathcal{C} \to \mathcal{A}$ and a *concretization function* $\gamma : \mathcal{A} \to \mathcal{C}$. The abstraction function assigns to each element $c \in \mathcal{C}$ the most precise element $\alpha(c)$ in $\mathcal{A}$ that represents c. Conversely, the concretization function assigns the element $\gamma(a)$ of $\mathcal{C}$ that contains exactly all concrete states that are represented by a to each abstract state $a \in \mathcal{A}$.

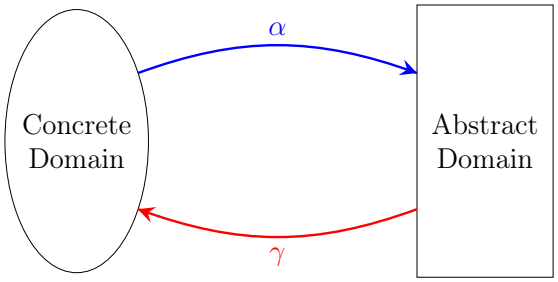


Figure 2.2: Domains in Abstract Interpretation.

The requirements of precision and over-approximation on α and γ have only been informally specified so far. They are enforced by demanding the functions to satisfy certain conditions described in the following definition.

Definition 2.3. Let $(L, \sqsubseteq)$ and $(P, \leqslant)$ be complete lattices. Two monotone functions $\alpha : L \to P$ and $\gamma : P \to L$ are said to form a *Galois connection*, written $L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} P$, if the following two conditions hold:

$$\alpha(\gamma(p)) \leqslant p$$
$$l \sqsubseteq \gamma(\alpha(l))$$

for all $p \in P$ and $l \in L$.

The definition is visualized in Figure 2.3. If $\alpha : L \to P$ and $\gamma : P \to L$ form a Galois connection, the former condition guarantees that α assigns the most precise possible element of P (depicted in the upper part of Figure 2.3) whereas the latter prohibits any unsound under-approximation (shown in the lower part of Figure 2.3).

Definition 2.4. A *representation function* $\beta : \Xi \to \mathcal{A}$ assigns to a single concrete state σ the least abstract state in $\mathcal{A}$ that over-approximates it.

Thus, its behavior can be described with the abstraction function α:

$$\beta(\sigma) = \alpha(\{\sigma\})$$

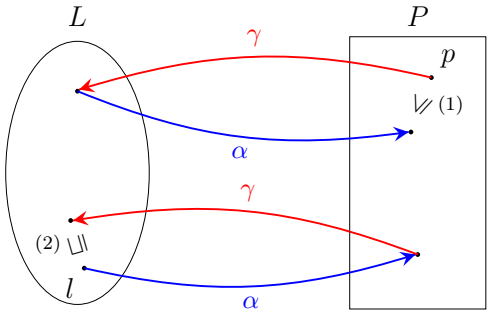


Figure 2.3: Visualization of the defining constraints of a Galois connection. The ordering at (1) guarantees precision, (2) provides soundness.

2.1.2 Reduced Products

Although the definition of the Galois connection introduces several restrictions to the abstraction and concretization functions to enforce soundness and precision, it is not guaranteed that the abstract domain does not contain redundant elements that are irrelevant for the approximation. There might exist two different abstract states representing the same set of concrete states. This is particularly of interest when combining abstract domains for more expressive analysis results. If the domains capture non-disjoint program properties, the cartesian product of them will contain such redundant states.

For example, if an interval analysis discovers that a program variable is bounded by $[-3, 7]$, and a parity analysis finds that the same variable has an even value in every execution, the combined state of both analyses describes the same set of concrete states as the state $([-2, 6], \texttt{even})$.

For a program analysis, it would be desirable to obtain the abstract state in the combined domain with the most precise component values that describes the set of program states that is represented by the analysis result. To formalize this requirement, the notion of *semantic reductions* is introduced in [Cousot and Cousot, 1979]:

Definition 2.5. Let $(L, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (P, \leqslant)$ be a Galois connection. With the *reduction operator* $\varsigma : P \to P$ defined by

$$\varsigma(p) = \bigsqcap \{p' \mid \gamma(p) = \gamma(p')\}$$

one obtains the *semantic reduction* $\varsigma[(P, \leqslant)]$ of $(P, \leqslant)$ as

$$\varsigma[(P, \leqslant)] = (\{\varsigma(p) \mid p \in P\}, \leqslant)$$

In the notation of the previous definition, $\varsigma[(P, \leqslant)]$ is a complete lattice and with γ restricted to the domain $\varsigma[(P, \leqslant)]$, $(L, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \varsigma[(P, \leqslant)]$ form a Galois connection. $\varsigma[(P, \leqslant)]$ contains exactly one representative of each set of mutually redundant abstract states in $(P, \leqslant)$.

Definition 2.6. Let $\mathcal{A}_1 = (A_1, \sqsubseteq_1)$, $\mathcal{A}_2 = (A_2, \sqsubseteq_2)$ be abstract domains with functions α_1, γ_1 and α_2, γ_2 such that $\mathcal{C} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \mathcal{A}_1$ and $\mathcal{C} \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \mathcal{A}_2$ form a Galois connection each. The complete lattice

$$\mathcal{A}_1 \times \mathcal{A}_2 = (A_1 \times A_2, \sqsubseteq)$$

with

$$(a_1, a_2) \sqsubseteq (a_3, a_4) \Leftrightarrow a_1 \sqsubseteq_1 a_3 \wedge a_2 \sqsubseteq_2 a_4$$

is called the *direct product* of $\mathcal{A}_1$ and $\mathcal{A}_2$.

In the same notation, with

$$\gamma((a_1, a_2)) = \gamma_1(a_1) \sqcap \gamma_2(a_2)$$
$$\alpha(c) = (\alpha_1(c), \alpha_2(c))$$

$\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}_1 \times \mathcal{A}_2)$ forms a Galois connection.

The previous definitions allow us to define a semantic combination operation for two complete lattices:

Definition 2.7. The complete lattice

$$\mathcal{A}_1 * \mathcal{A}_2 = \varsigma[\mathcal{A}_1 \times \mathcal{A}_2]$$

is said to be the *reduced product* of $\mathcal{A}_1$ and $\mathcal{A}_2$.

The reduced product of two abstract domains combines the information from the domains and avoids the imprecision introduced by redundant abstract states. The results of an analysis that uses the reduced product of two domains are often more precise than the results that can be achieved by iteratively analyzing with the component domains separately in any order.

Combining two existing analyses in a reduced product is a highly non-trivial task that typically is only possible by creating a new combined abstract domain entirely from scratch as pointed out in [Cousot and Cousot, 1979].

2.1.3 Transformer Functions

The execution of program statements τ modifies the state of the system. This is modeled in the concrete domain with *concrete transformers* $post[\tau] : \Xi \to \Xi$ for every possible program statement. Concrete transformers for single concrete states can be raised to the concrete domain by element-wise application:

$$post[\tau] : \mathcal{C} \to \mathcal{C},\ c \mapsto \{post[\tau](\sigma) \mid \sigma \in c\}$$

For an effective abstract-interpretation-based program analysis, a corresponding *abstract transformer* $\widetilde{post}[\tau] : \mathcal{A} \to \mathcal{A}$ that over-approximates the semantics of τ in the abstract domain is necessary.

Definition 2.8. The *best abstract transformer* $\widehat{post}[\tau] : \mathcal{A} \to \mathcal{A}$ maps an abstract value $a \in \mathcal{A}$ to the least abstract value $\widehat{post}[\tau](a)$ that over-approximates any concrete state that is reachable by executing the statement τ at a concrete state that is represented by a (visualized in Figure 2.4):

$$\widehat{post}[\tau] = \alpha \circ post[\tau] \circ \gamma$$

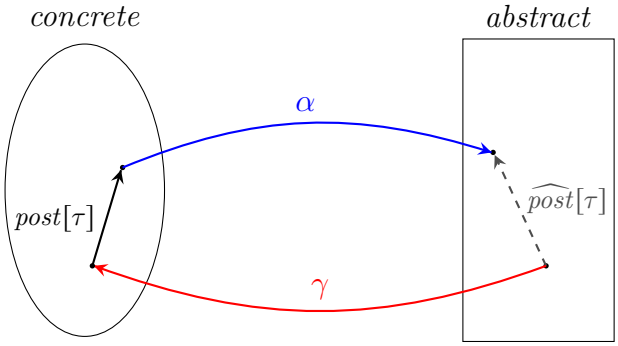


Figure 2.4: Definition of the best abstract transformer $\widehat{post}[\tau]$.

However, this definition of the best abstract transformer cannot be used as an algorithm for its systematic computation as the concretization function can easily generate infinite or infeasibly large sets of concrete states for which the element-wise concrete transformer is not computable.

Manually implementing best abstract transformers is a tedious and error-prone task which has to be repeated for every possible program statement when implementing a new abstract domain. An obvious measure for reducing the necessary effort is to implement only simplified over-approximations of the best abstract transformer at the cost of a potential loss of precision.

With a formalization of the abstract domain and the abstract transformers for all program statements, the *abstract semantics* of a program can be defined as a solution of an equation of the form

$$X = F(X)$$

where $X \in \mathcal{A}^k$ contains an abstract value for each abstraction point of the program and $F : \mathcal{A}^k \to \mathcal{A}^k$ consists of applications of abstract transformers (cf. [Cousot and Cousot, 1977]). Such solutions can be computed by finding a fixed point of F.

2.2 Satisfiability Modulo Theories

Investigating the satisfiability of logical formulas is an important field of research in computer science. It plays a substantial role in computability and complexity theory, and many other disciplines of computer science heavily depend on the existence of

efficient algorithms for checking satisfiability (like e.g. artificial intelligence, computer assisted proof generation and program verification).

Typical examples for logical systems that are subject to satisfiability checking are *propositional logic* and the more expressive *first-order logic*. Although useful for some applications, those logics have a significant disadvantage for many potential uses: they do not provide predicates with a fixed semantics. Consider the following first-order formula as an example:

$$x < y \wedge x = y + 1$$

Simple first-order satisfiability checkers would just see $[\cdot] < [\cdot]$ and $[\cdot] = [\cdot] + 1$ as predicates over variables and would try to come up with an interpretation for these predicates such that the formula is true. This is not a desirable behavior if one wants to check whether there is a mapping of x and y to integers such that they fulfill both parts of the formula with the typical meaning of integer comparison, addition and equality.

This consideration leads to the notion of *Satisfiability Modulo Theories* (SMT) [Barrett et al., 2009]: Certain predicates in the formula are no longer uninterpreted but evaluated in a way that is specified by the chosen theory. This allows for example to check satisfiability of formulas with respect to integer arithmetics, array operations or fixed-width bit vector arithmetics.

For this thesis, a subclass of the last mentioned example, namely *quantifier-free bit vector arithmetics* (QFBV) is the most frequently used theory. As the thesis only utilizes SMT for checking the satisfiability of certain formulas, a solver is used as a black box for model generation. The decision procedure for this instantiation of SMT has been proven to be NEXPTIME-complete [Kovásznai et al., 2012].

2.3 Symbolic Abstract Interpretation

The concept of *Symbolic Abstract Interpretation* was first introduced in [Reps et al., 2004] and further developed and summarized in [Thakur, 2014] as an extension to the Abstract Interpretation framework (section 2.1). It adds an additional symbolic layer between concrete and abstract domains: In this *symbolic domain* $\mathcal{S}$, sets of concrete program states are represented by formulas of some given logic $\mathcal{L}$, e.g. an instance of SMT. If $\mathcal{L}$ is expressive enough, the concrete transformer functions $post[\tau]$ can be represented by formulas φ_τ capturing their respective behavior independently of the abstract domain.

For the following definitions, it is convenient to assume the existence of a function $[\![\cdot]\!] : \mathcal{S} \to \mathcal{C}$ that maps a formula to the set of concrete states that it represents.

Corresponding to the abstraction and concretization functions α and γ, one can define a *symbolic abstraction function* $\widehat{\alpha} : \mathcal{S} \to \mathcal{A}$ and a *symbolic concretization function* $\widehat{\gamma} : \mathcal{A} \to \mathcal{S}$. $\widehat{\alpha}$ maps formulas $\psi \in \mathcal{S}$ to the least abstract value $\widehat{\alpha}(\psi)$ that over-approximates the set of concrete values that ψ represents and $\widehat{\gamma}$ maps abstract values a to a formula $\widehat{\gamma}(a)$ in $\mathcal{S}$ that represents the same set of concrete states as a (cf. Figure 2.5).

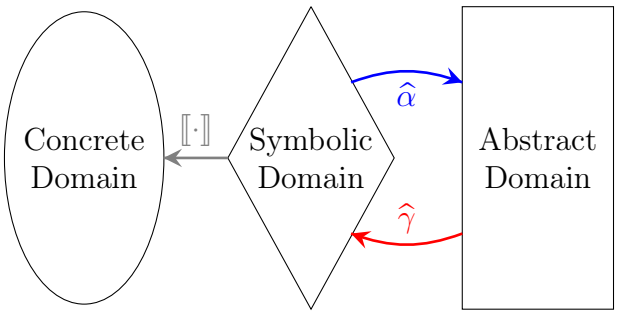


Figure 2.5: Domains in Symbolic Abstract Interpretation.

The symbolic abstraction and concretization functions can therefore be defined in terms of their non-symbolic counterparts as follows:

$$\widehat{\alpha}(\psi) = \alpha([\![\psi]\!])$$
$$\gamma(a) = [\![\widehat{\gamma}(a)]\!]$$

The effect of a program statement τ can be specified as the program state after the execution of τ in terms of the program state before. To specify such a relation in SMT formulas, two disjoint sets Σ_{pre} and Σ_{post} of SMT variables are necessary, one representing the state before the execution of the statement and one for the state afterwards. In the following, for a program variable `x`, the corresponding variable in Σ_{post} that represents its state after the execution is denoted by a primed variable x' with the same name whereas the state before execution of the statement is represented by the unprimed version x of the variable in Σ_{pre}. It is convenient for the formalization to parameterize the transformer formulas $\varphi_\tau^{\Sigma_{pre},\Sigma_{post}}$ by the sets Σ_{pre} and Σ_{post} and similarly the elements ψ^Σ of the symbolic domain by the set Σ of SMT variables that are used to represent the state of program variables. In this notation, superscripts of $\widehat{\alpha}$ and $\widehat{\gamma}$ describe the set of SMT variables on which the corresponding function operates on.

For example, the semantics of the assignment

```
x = y + z;
```

could be captured with the formula

$$(x' = y + z) \wedge (y' = y) \wedge (z' = z).$$

A symbolic transformer formula $\varphi_\tau^{\Sigma_1,\Sigma_2}$ can be applied to an element ψ^Σ of the symbolic domain by forming the conjunction $\varphi_\tau^{\Sigma_{pre},\Sigma_{post}} \wedge \psi^{\Sigma_{pre}}$ of the two formulas and interpreting the variables from Σ_{post} as the significant variables constraining the represented concrete states. Hence, two symbolic transformers $\varphi_{\tau_1}^{\Sigma_1,\Sigma_2}$ and $\varphi_{\tau_2}^{\Sigma_3,\Sigma_4}$ can be concatenated by forming the conjunction of the two formulas $\varphi_{\tau_1}^{\Sigma_{pre},\Sigma_{int}}$ and $\varphi_{\tau_2}^{\Sigma_{int},\Sigma_{post}}$ that share a set Σ_{int} of intermediate variables for the state after the first statement and before the second statement:

$$\begin{aligned}\psi^{\Sigma_{pre}} \wedge \varphi_{\tau_{1,2}}^{\Sigma_{pre},\Sigma_{post}} &\equiv \psi^{\Sigma_{pre}} \wedge (\varphi_{\tau_1}^{\Sigma_{pre},\Sigma_{int}} \wedge \varphi_{\tau_2}^{\Sigma_{int},\Sigma_{post}}) \\ &\equiv (\psi^{\Sigma_{pre}} \wedge \varphi_{\tau_1}^{\Sigma_{pre},\Sigma_{int}}) \wedge \varphi_{\tau_2}^{\Sigma_{int},\Sigma_{post}}\end{aligned}$$

With this technique, arbitrary loop free program fragments can be represented by a single transformer formula that avoids unnecessary loss of precision induced by conversions to the abstract domain in between.

These definitions allow a characterization of the best abstract transformer $\widehat{post}[\tau]$ similar to Definition 2.8 (cf. Figure 2.6): The best abstract transformer function for an arbitrary transformer in an arbitrary abstract domain can be computed by applying the symbolic abstraction function to the conjunction of the formula representing the transformer and the symbolic concretization of the input abstract value:

$$\widehat{post}[\tau](a) = \widehat{\alpha}^{\Sigma_{post}}(\varphi_{\tau}^{\Sigma_{pre},\Sigma_{post}} \wedge \widehat{\gamma}^{\Sigma_{pre}}(a))$$

Note that the above characterization does not require the abstract domains of the value before and after the transition to be identical. By using the symbolic concretization function of the input abstract domain and the symbolic abstraction function of the output domain, this construction can be used to convert information between abstract domains. Thus, a static analyzer can choose different abstract domains for different program points, possibly guided by human input.

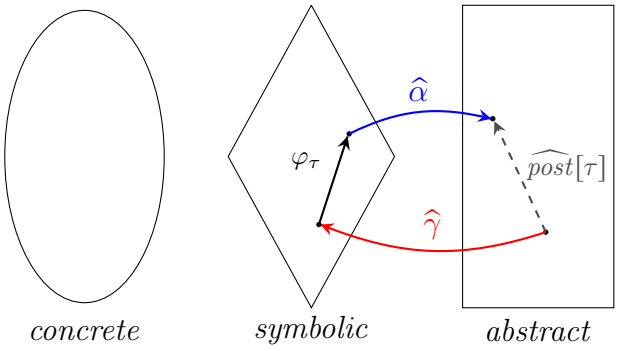


Figure 2.6: Best abstract transformer in Symbolic Abstract Interpretation.

2.3.1 Computing $\widehat{\alpha}$

A major advantage of Symbolic Abstraction is that, in contrast to standard Abstract Interpretation, the described definition of the best abstract transformer is effectively computable. Designing the formulas φ_τ for the transformers and the symbolic concretization function $\widehat{\gamma}$ are straightforward tasks if the desired semantics of the analyzed language and the abstract domain are known (examples for such transformer formulas and symbolic concretization functions can be found in section 4.2 and section 4.4). The

only remaining challenge for precise abstract transformers is a general implementation of the symbolic abstraction function $\widehat{\alpha}$.

In the following, two algorithms for computing $\widehat{\alpha}$ independently of the abstract domain are described: a unilateral approach that approximates the final result from below until a sound result is reached and a bilateral approach that tracks lower and upper bounds of the best possible result and refines them until they are equal. The algorithms rely on an appropriate logics solver that is able to generate models (i.e. satisfying variable assignments) for satisfiable formulas.

For abstract domains with finite height, both algorithms are guaranteed to terminate with the most precise result that is possible. In the case of domains with infinite height, none of the algorithms is guaranteed to terminate. However, the latter of the presented algorithms can still give non-trivial and sound over-approximations of the correct result if it is stopped at any point in time.

Unilateral Algorithm

The first approach for a general computation of $\widehat{\alpha}$ is the unilateral algorithm as presented in [Reps et al., 2004]. The algorithm denoted here as $\widehat{\alpha}^{\uparrow}(\varphi)$ (Algorithm 2.1) is the version described in [Thakur, 2014] that follows the same concept. It requires the abstract domain to provide

- a lattice join $\sqcup : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$,
- a symbolic concretization function $\widehat{\gamma} : \mathcal{A} \to \mathcal{S}$ and
- a representation function $\beta : \Xi \to \mathcal{A}$.

For finding the most precise representation of a formula φ in the abstract domain, the algorithm keeps a lower bound *lower* of the correct result and refines it until it is no longer an under-approximation of the result. Starting with the $\bot$ value of the abstract domain, the first step is to ask the solver for a model for the conjunction of φ with the negation $\neg\widehat{\gamma}(\mathit{lower})$ of the symbolic concretization of the current lower bound. That is a model of a concrete state that is represented by the input formula φ but not by the current lower bound. If no such model exists, i.e. the conjunction is unsatisfiable, every concrete state that is represented by φ is also captured by *lower* and therefore *lower* soundly abstracts the values represented by φ.

If a model S is found, it corresponds to a concrete state σ that is not represented by *lower* but that should be captured by $\widehat{\alpha}(\varphi)$. It is beneficial to identify the model S with the concrete state σ that it represents. Thus, one can get a new lower bound that represents everything captured by *lower* and by S by taking the join of *lower* and the representative $\beta(S)$ of S in the abstract domain. With this new lower bound, the algorithm repeats the performed steps until no model for a concrete state from $[\![\varphi]\!]$, that is not represented by the lower bound, can be found anymore.

Algorithm 2.1: $\widehat{\alpha}^{\uparrow}(\varphi)$

```
1  lower ← ⊥
2  while true do
3    S ← FindModel(φ ∧ ¬γ̂(lower))
4    if S is None then
5      break
6    else
7      lower ← lower ⊔ β(S)
8  done
9  ans ← lower
10 return ans
```

Algorithm 2.2: $\widehat{\alpha}^{\uparrow}_{inc}(\varphi)$

```
1  lower ← ⊥
2  ψ ← φ
3  while true do
4    ψ ← ψ ∧ ¬γ̂(lower)
5    S ← FindModel(ψ)
6    if S is None then
7      break
8    else
9      lower ← lower ⊔ β(S)
10 done
11 ans ← lower
12 return ans
```

The $\widehat{\alpha}^{\uparrow}(\varphi)$ algorithm from [Thakur, 2014] differs slightly from the version from [Reps et al., 2004]. Reps et al. update the input formula in every loop iteration to be the conjunction of the previous formula with the negated symbolic concretization of the current lower bound. In consequence, the calls to the solver for this version contain not only subexpressions for the current lower bound but also for every lower bound that has been considered in previous loop iterations. As a newly obtained lower bound $lower'$ always subsumes the previous lower bound $lower$, adding the negated symbolic concretizations of all so far encountered lower bounds is equivalent to adding only the subexpression for the newest lower bound to the solver call:

$$\begin{aligned} lower \sqsubseteq lower' &\Rightarrow \widehat{\gamma}(lower) \rightarrow \widehat{\gamma}(lower') \\ &\Rightarrow \neg\widehat{\gamma}(lower') \rightarrow \neg\widehat{\gamma}(lower) \\ &\Rightarrow (\neg\widehat{\gamma}(lower') \wedge \neg\widehat{\gamma}(lower)) \equiv \neg\widehat{\gamma}(lower') \end{aligned}$$

Despite having a larger formula as input for the SMT solver, the fact that previous clauses are kept as assumptions for the solver allows the solver to reuse information that has been generated for one call to the solver in the following calls without having to recompute it every time. This can potentially improve the actual runtime of the algorithm significantly.

A version of the algorithm that is similar to the one presented in [Reps et al., 2004] in its use with conjuncts for previous lower bounds is shown as $\widehat{\alpha}^{\uparrow}_{inc}(\varphi)$ in Algorithm 2.2. An example run of this algorithm is visualized in Figure 2.7.

Bilateral Algorithm

Depending on the model generation of the SMT solver, the unilateral algorithm can involve checking for every element in a maximal ascending chain of the abstract domain whether it subsumes the concrete values that should be represented for every abstract transformer.

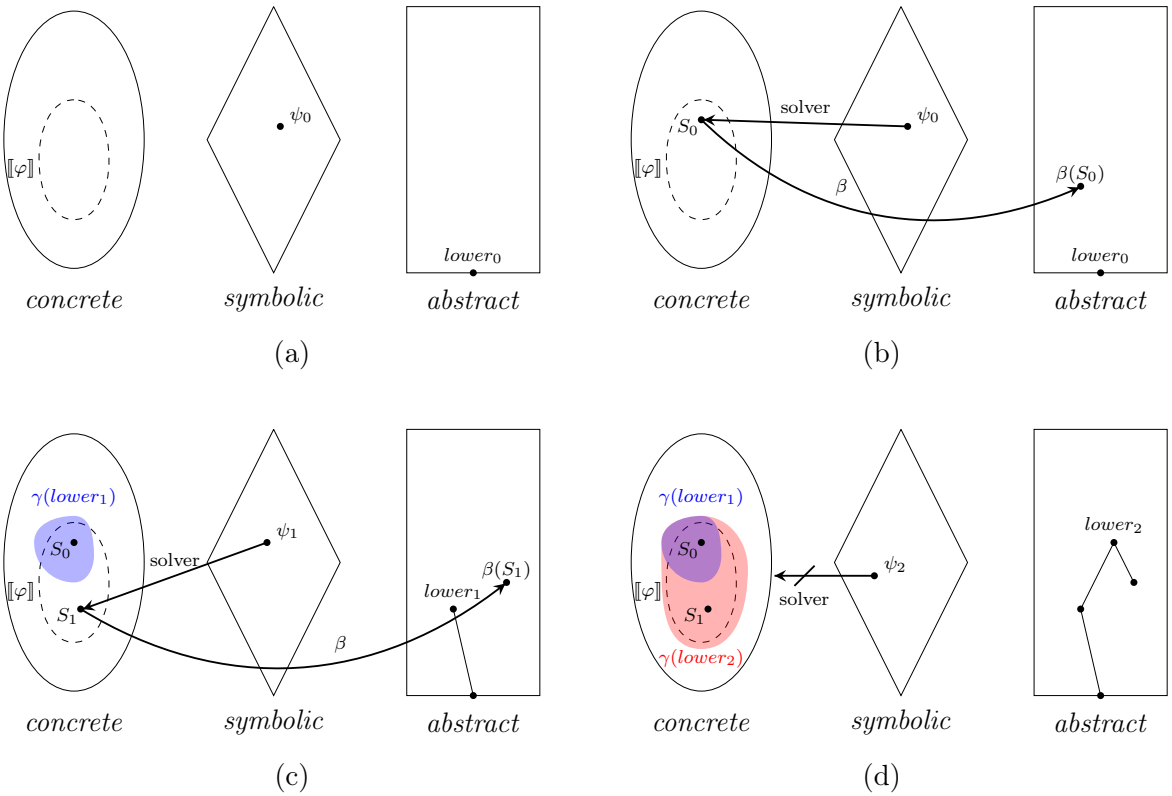


Figure 2.7: Example run of the unilateral $\widehat{\alpha}^{\uparrow}_{inc}(\varphi)$ algorithm.
Initially (a), $lower_0$ is $\bot$ and ψ_0 is initialized as $\varphi \wedge \neg\texttt{false}$. None of the concrete states represented by φ (the dashed ellipsis) is captured by $lower_0$. Next (b), the solver finds a model S_0 for ψ_0 with the representative $\beta(S_0)$. In the following step (c), $lower_1$ is set to the join of $lower_0$ and $\beta(S_0)$ (here: $\beta(S_0)$ as $lower_0 = \bot$). $lower_1$ covers only a part of the concrete states in $[\![\varphi]\!]$ (blue area), therefore the solver finds a model S_1 for $\psi_1 := \psi_0 \wedge \neg\widehat{\gamma}(lower_1)$. Finally (d), with the new lower bound $lower_2 = lower_1 \sqcup \beta(S_1)$, the solver finds that $\psi_2 := \psi_1 \wedge \neg\widehat{\gamma}(lower_2)$ is unsatisfiable, hence $lower_2$ over-approximates the concrete states represented by φ (red area).

The bilateral algorithm for symbolic abstraction presented in [Thakur, 2014] introduces a notion to improve the performance of symbolic abstraction in such cases (here described as $\widehat{\alpha}^{\updownarrow}(\varphi)$, cf. Algorithm 2.3).

The following definition formalizes a necessary concept for this algorithm:

Definition 2.9. A function $\texttt{AbsCons} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is said to be an *abstract consequence operation* if for all $l, u \in \mathcal{A}$ the following implication holds:

$$l \sqsubsetneq u \Rightarrow l \sqsubseteq \texttt{AbsCons}(l, u) \wedge \texttt{AbsCons}(l, u) \not\sqsupseteq u$$

An abstract consequence operation has to provide for its arguments l, u an abstract value that is greater than or equal to l and neither greater than nor equal to u. Note

that this guarantees that the meet of the abstract consequence and u lies "between" l and u (including l and excluding u). If the operation provides abstract values such that maximal ascending chains from l to the value and from the value to u are of similar length, it can be used to determine the symbolic abstraction of a formula in a binary-search-like manner, effectively reducing the runtime in comparison to the unilateral algorithm.

Furthermore, the abstract consequence operation can be implemented to yield abstract values with possibly less complex symbolic concretized formulas, potentially reducing the runtime of the calls to the solver and therefore improving the overall performance.

With this concept, all operations necessary for the bilateral algorithm are available: In addition to the operations that a domain has to support for the unilateral algorithm,

- a lattice meet $\sqcap : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ and
- a valid abstract consequence operation `AbsCons` : $\mathcal{A} \times \mathcal{A} \to \mathcal{A}$

is needed.

Algorithm 2.3: $\widehat{\alpha}^{\updownarrow}(\varphi)$

```
1  upper ← ⊤
2  lower ← ⊥
3  while lower ≠ upper do
4    p ← AbsCons(lower, upper)
5    S ← FindModel(φ ∧ ¬γ̂(p))
6    if S is None then
7      upper ← upper ⊓ p
8    else
9      lower ← lower ⊔ β(S)
10 done
11 ans ← upper
12 return ans
```

The bilateral algorithm keeps upper and lower bounds *upper*, *lower* of the precise result of the symbolic abstraction function. It starts with the trivial upper and lower bounds $\top$ and $\bot$, respectively. With the abstract consequence operation, an abstract value p "between" *lower* and *upper* is generated. The solver is called to provide a model for the conjunction of the input formula φ with the negated symbolic concretization $\widehat{\gamma}(p)$ of p. If a model S is found, it represents a concrete state that is not captured by *lower*, hence $lower \sqcup \beta(S)$ is also a valid lower bound of the correct result and *lower* is updated accordingly. Otherwise, if there is no such model, p is already an upper bound of the precise result and so is the meet of *upper* and p.

Either way, one of the bounds can be refined and the algorithm can start its next iteration if the bounds are not yet identical.

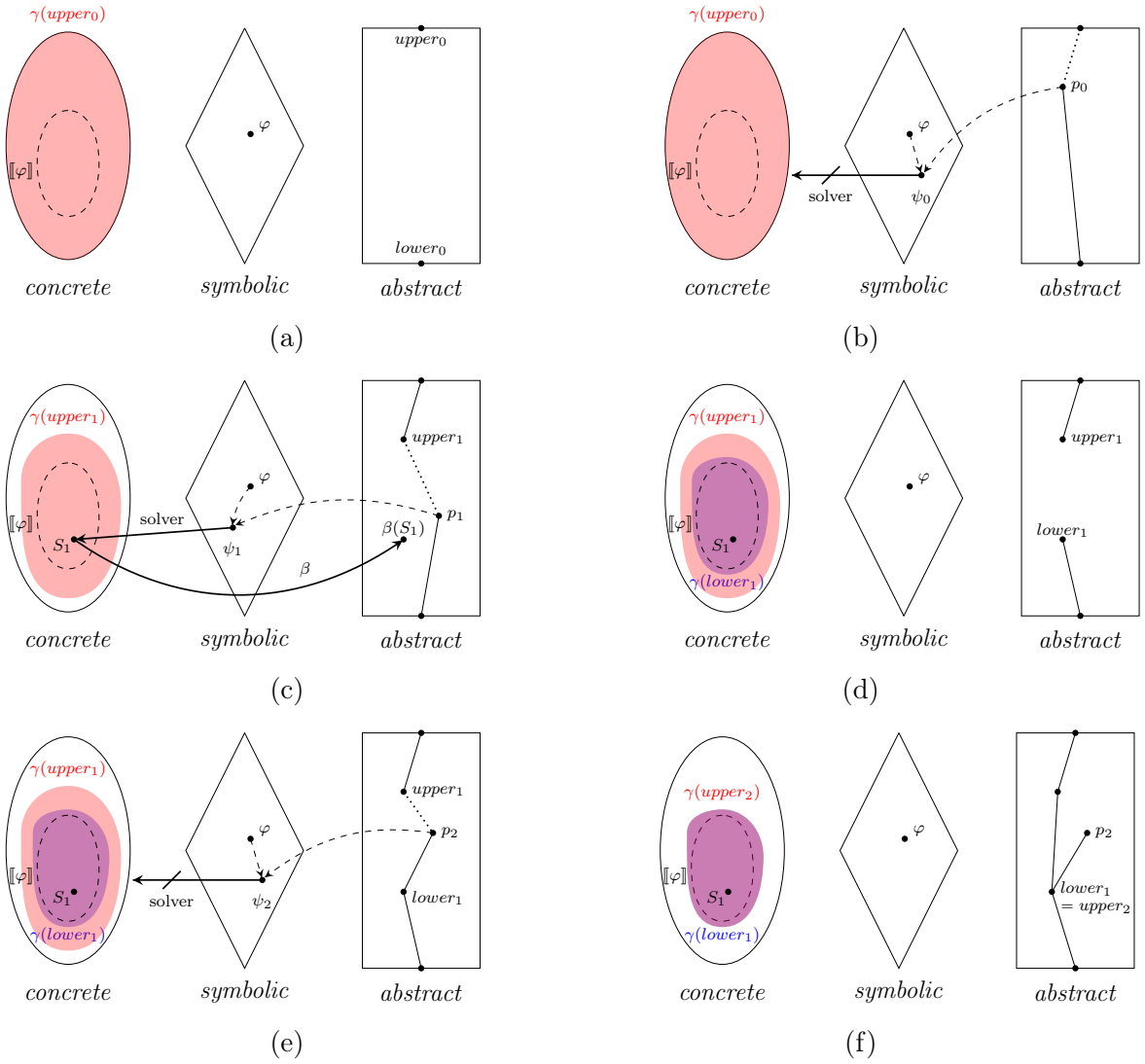


Figure 2.8: Example run of the bilateral $\widehat{\alpha}^{\updownarrow}(\varphi)$ algorithm.
Initially (a), $lower_0$ ($upper_0$) is defined as $\bot$ ($\top$). Hence, $\gamma(lower_0)$ is the empty set whereas $\gamma(upper_0)$ subsumes the set $\mathcal{C}$ of all program states (red area). The input φ represents the dashed subset of $\mathcal{C}$. First (b), an abstract consequence p_0 of $lower_0$ and $upper_0$ is computed and a solver call is issued for $\psi_0 := \varphi \wedge \neg\widehat{\gamma}(p_0)$. Here, the solver finds no model, so in the next step (c), the upper bound is refined to $upper_1 := upper_0 \sqcap p_0 = p_0$. Again, an abstract consequence p_1 of $lower_0$ and $upper_1$ is generated and the formula $\psi_1 := \varphi \wedge \neg\widehat{\gamma}(p_1)$ is used as an input for the solver. The solver discovers a model S_1 whose representative $\beta(S_1)$ refines the lower bound. Now (d), $lower_1 = lower_0 \sqcup \beta(S_1)$ captures a non-empty subset of the concrete domain (blue area). Once again (e), a solver call for the conjunction ψ_2 of φ and the negated symbolic concretization of an abstract consequence p_2 of the bounds is issued. As no model is found, the $p_2 \sqcap upper_1$ is used as the new upper bound $upper_2$ in (f). Now, both bounds have the same value, with which the algorithm terminates.

Notice that always returning the first argument is a valid behavior for an abstract consequence operation. With such an abstract consequence operation, the bilateral algorithm degenerates to perform the same steps as the unilateral algorithm: First, only the lower bound is refined until it reaches the correct result, then the solver is called for an unsatisfiable formula and the upper bound is refined to the value of the lower bound. However, there is no obvious way of modifying the bilateral algorithm to benefit in runtime from previous solver queries like it is done in the $\widehat{\alpha}^{\uparrow}_{inc}(\varphi)$ algorithm.

One further benefit of the bilateral algorithm is that the upper bound that it keeps throughout the application of the algorithm always is a sound over-approximation of the precise result of the symbolic abstraction. Hence, the algorithm can be modified to restrict the overall time for calls to the solver and to still provide sound, non-trivial results in cases where the time is not sufficient to generate precise results. In such a case, the loop of the algorithm can be broken and the upper bound is returned.

On the other side, one can observe that — compared to the unilateral algorithm — the bilateral algorithm can issue significantly more calls to the SMT solver if one starts the algorithm with an already valid over-approximation of the result as a lower bound. The bilateral algorithm has to refine the upper bound until it reaches the lower bound whereas the unilateral algorithm only needs to perform a single unsatisfiable solver call.

2.3.2 Reduced Products in Symbolic Abstract Interpretation

Symbolic Abstract Interpretation allows a flexible and modular domain design. Small and easy to specify abstract domains can be combined without additional effort in a reduced product:

Given two abstract domains $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ with operations $\sqcup_A$, $\sqcap_A$, $\widehat{\gamma}_A$, β_A and $\sqcup_B$, $\sqcap_B$, $\widehat{\gamma}_B$, β_B, respectively, for the reduced product $(A, \sqsubseteq_A) * (B, \sqsubseteq_B)$, we define:

$$\begin{aligned} A * B &:= A \times B \\ (a, b) \sqcup_{A*B} (c, d) &:= (a \sqcup_A c, b \sqcup_B d) \\ (a, b) \sqcap_{A*B} (c, d) &:= (a \sqcap_A c, b \sqcap_B d) \\ \widehat{\gamma}_{A*B}((a, b)) &:= \widehat{\gamma}_A(a) \wedge \widehat{\gamma}_B(b) \\ \beta_{A*B}(\sigma) &:= (\beta_A(\sigma), \beta_B(\sigma)). \end{aligned}$$

The new product domain is the cartesian product of the combined domains. Join, meet and representation function are applied component-wise and the new symbolic concretization function returns the conjunction of the results of the symbolic concretization functions of the combined domains.

As the symbolic abstraction algorithm is guaranteed to return the most precise element of the cartesian product that over-approximates the input, this construction yields the same results as applying the reduction operator ς to a direct product of the contributing domains.

This construction can be lifted to an arbitrary number of combined abstract domains in a straightforward way.

3 Used Frameworks

The software developed in the thesis uses and builds upon existing frameworks that are described in this chapter.

3.1 LLVM

The goal of the LLVM project [Lattner and Adve, 2004] is providing a modular compiler framework that uses modern techniques to support expressive code analyses and optimizations. For this purpose, LLVM defines a low-level intermediate representation (LLVM IR) based on a control flow graph in Single Static Assignment (SSA) form [Cytron et al., 1991] with high-level type information.

In practice, this means that the program representation is a directed graph with basic blocks as nodes. These basic blocks represent sequences of simple instructions that are all executed unconditionally if the basic block is executed. The instructions can be divided into two categories: the ϕ-nodes and the non-ϕ instructions. The latter ones include typical assembly-like instructions, e.g. for arithmetical operations or branches. Their result only depends on the values of their operands. In contrast to that, the former ones are conditional copy-operations depending on the previously executed block. A basic block always consists of a (possibly empty) sequence of ϕ-nodes (the "ϕ-part") and a sequence of other instructions ending in a terminator instruction that determines the basic block to be executed afterwards (together the "non-ϕ-part").

Most LLVM instructions generate a result from their operands. For reasons of disambiguation, the representations of instruction results will be referred to as *SSA-variables* or just *variables* in the rest of the thesis. These variables have a static type and — following the lines of the definition of the SSA representation — are defined exactly once in a function and each use of an SSA-variable is dominated by its definition. The possible results of instructions in concrete runs of the program will be called *values* throughout the rest of the thesis. This is in contrast to the terminology used in the LLVM source code, where the class `Value` is a superclass of most of the CFG constructs (including basic blocks and instructions).

The choice of a program representation in SSA form simplifies the implementation of program optimizations. Its benefits for this thesis are explained in section 4.2.

The semantics of the LLVM IR instructions is designed to follow the semantics of C and C++ in most aspects; nevertheless its structure is simple enough to be compiled to efficient machine code for different hardware platforms.

LLVM front ends are available for many commonly used programming languages like C/C++[1], Haskell[2] and Rust[3] which makes optimizations on LLVM IR available for all those languages without extra effort.

Optimizations in the LLVM world are typically implemented as passes that can easily be run on the IR code or directly in the compilation process. LLVM comes with a considerable range of optimization and analysis passes whose results can be used in newly created passes.

Existing passes that are used and discussed in this thesis include[4]:

SCCP Sparse Conditional Constant Propagation, a program transformation proposed in [Wegman and Zadeck, 1991]. It combines the folding and propagation of variables with a constant value at compile time with the elimination of unreachable code. As constant values often imply constant branching conditions which can cause unreachable code and the removal of unreachable code in turn can make constant values discoverable at compile time, this is a powerful program optimization.

The LLVM implementation handles most kinds of LLVM values, including floating point values. An inter-procedural version that considers constants from call sites is also available.

SimplifyCFG A pass that simplifies the control flow graph syntactically by removing obviously unreachable basic blocks, eliminating trivial ϕ-node and merging basic blocks if possible.

DCE The LLVM Dead Code Elimination removes instructions whose results are not used in further computations. Together with SimplifyCFG, it provides a powerful set of clean up transformations.

GVN LLVM also comes with a version of the global value numbering analysis presented in [Kildall, 1973] with sophisticated transformations based on their results. It does not only eliminate redundant computations and common subexpressions, but also removes dead loads. New ϕ-nodes can be introduced to propagate results from a non-dominating basic block to remove redundant computations more effectively.

3.2 Z3

Z3 [de Moura and Bjørner, 2008] is an efficient and advanced SMT solver intended for software verification and software analysis, developed by Microsoft Research. It

[1] `http://clang.llvm.org/`
[2] `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/Backends/LLVM`
[3] `http://www.rust-lang.org/`
[4] a complete list of LLVM passes can be found at `http://llvm.org/docs/Passes.html`

supports several theories that can be used for modeling software behavior such as bit vectors (for machine integer arithmetics) and arrays (for memory). There is also basic support for floating point types that can be used for modeling floating point arithmetics.

Due to the nature of the implemented *conflict-driven clause learning* algorithm [Biere et al., 2009], an instance of the Z3 solver learns additional lemmas that help in the process of proving or disproving satisfiability of the formula during a satisfiability check. Hence, it can be beneficial to reuse solver instances for formulas that share significant common conjuncts as the lemmas from previous runs can be reused avoiding unnecessary slowdowns because of lemma recomputation.

The key arguments for the use of Z3 are its relatively good performance and the C++ interface that reveals most of the relevant functionality in a convenient way.

4 Sprattus

Sprattus (Static Program Analysis ToolkiT Using Satisfiability) is a framework for static analysis of programs in the LLVM intermediate representation. It emerged from a project for the *Static Program Analysis* course in WS 2014/15 at Saarland University.

Major advantages of the toolkit are its flexibility in the design of new abstract domains and the simple generation of reduced products of arbitrary combinations of domains. Based on algorithms for symbolic abstraction described in section 2.3 and powered by Z3, a domain designer only needs to specify a few essential operations on single abstract states as well as their semantics in SMT formulas. Providing abstract transformers for the numerous LLVM instructions is not necessary.

The toolkit enables precise results even for domains with very limited expressiveness by supporting abstract transformers for large program fragments and only abstracting at few necessary program points, e.g. for handling loops. This fact makes Sprattus interesting for program analysis purposes as well as for applications in software verification.

On the other side, the performance of Sprattus is limited by the immense worst-case complexity of the approach given by the interaction with the SMT solver. Precisely modeling memory behavior or floating point operations can increase the expected analysis time heavily. Furthermore, LLVM vector instructions as well as some other operations are not supported in Sprattus.

The Sprattus framework provides several features that are necessary for a static program analysis based on symbolic abstraction:

- formula generation for constructing symbolic transformers for LLVM IR,
- implementations of the generic symbolic abstraction function,
- an engine for computing fixed points of the program semantics,
- an interface for specifying abstract domains, and
- a library of abstract domains that can be used directly or as a base for the implementation of new domains.

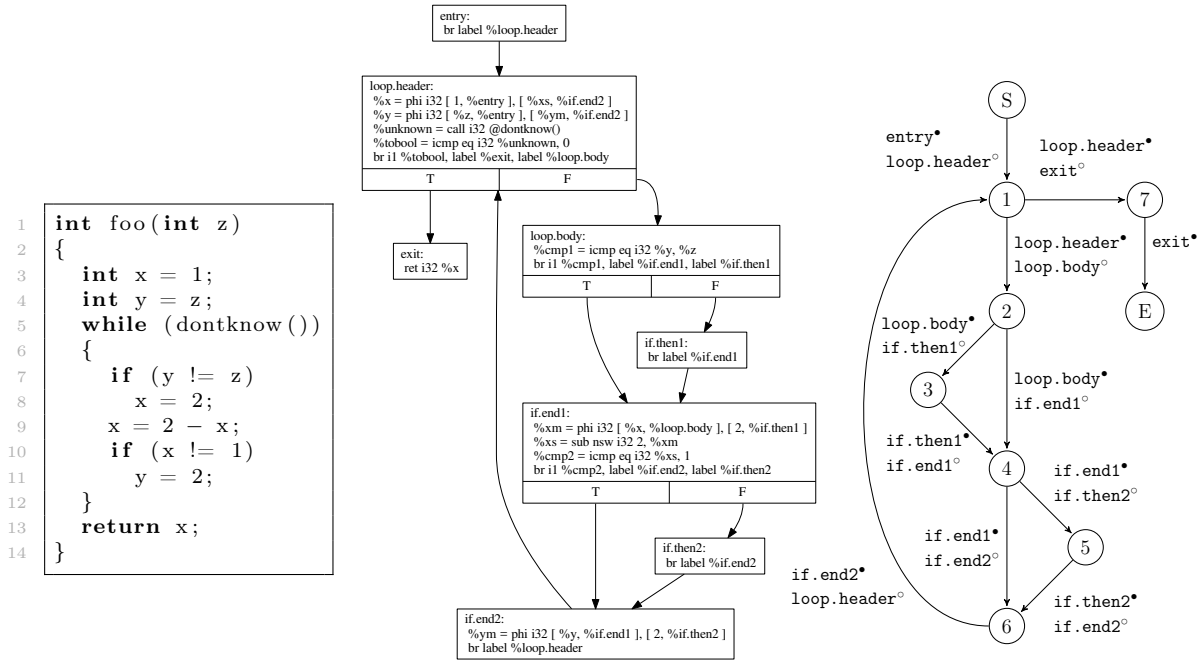
These components and their interaction are described in the following sections.

4.1 Program Representation

To be analyzed in the SPRATTUS framework, an input program in a high-level programming language like C or C++ has to be translated into the LLVM intermediate representation in SSA form.

LLVM IR is usually represented graphically as a *control flow graph* (CFG) with basic blocks as nodes and (directed) edges representing possible control flow between them.

In static analysis, it is more convenient to assign instructions to edges instead of nodes (so-called edge effects). Since ϕ-nodes behave differently depending on the incoming edge, we have chosen an interpretation in which an edge from a basic block `a` to a basic block `b` represents the non-ϕ-instructions of `a` and the ϕ-nodes of `b`. In this scenario, the ϕ-nodes of an edge are just unconditional copies of the values corresponding to a transition from the source basic block of the edge. To have the non-ϕ-instructions of function exit blocks included in control flow edges, it is necessary to add an artificial `EXIT` node that contains no instructions and that is a successor of all nodes that correspond to basic blocks without regular successors.



(a) input program (b) LLVM IR CFG (c) edge-based CFG

Figure 4.1: Representations of an example program, which is taken from [Click and Cooper, 1995]. (a) Shows the program in the C programming language, (b) is a graphical depiction of the LLVM IR that the CLANG compiler generates from the input. The edge-based CFG in (c) shows how the IR is interpreted for SPRATTUS. Here, `bb`$^\bullet$ represents the instructions from basic block `bb` that are no ϕ-nodes, `bb`$^\circ$ represents the ϕ-nodes of `bb`.

This interpretation of the LLVM IR is not made explicit in SPRATTUS, however it substantially determines the structure of the symbolic transformer formulas as described in section 4.2.

These three program representations — source code in a high-level language, a graphical representation of the LLVM IR and the interpretation as a control flow graph with instructions assigned to edges — are displayed for an example program in Figure 4.1.

4.2 Formula Construction

In order to analyze a program using symbolic abstract interpretation, it is necessary to convert all the program parts to SMT formulas capturing their respective semantics. The size of the program fragments that are translated into a single formula influences the precision of the program analysis. With larger fragments, longer subsequent parts of the program are covered by a single abstract transformer. Consequently, results are abstracted at fewer program points and therefore potentially more precise.

For example, the control flow graph depicted in Figure 4.2 contains two nodes S and 3 that include a conditional branch influencing the flow of control in the function. If the conditions for both branches are equal, a best abstract transformer for the complete function yields the information that no execution can possibly take a path through the function that contains the nodes 1 and 5 (and similarly for 2 and 4). An analysis that only uses abstract transformers for single control flow edges cannot give this result without a very expressive and specifically-crafted abstract domain.

Despite producing fewer solver calls with larger fragments, those that are issued are more complex and might result in a worse overall performance than smaller fragments.

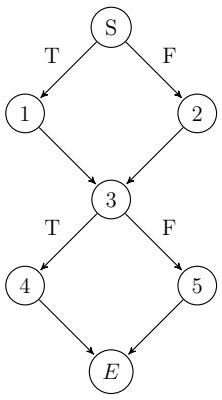


Figure 4.2: Example CFG for which larger fragments improve analysis precision.

4.2.1 Basic-Block-Level Construction

As basic blocks are always executed as a whole and they are directly accessible in the IR, it is tempting to use them as smallest elements for atomically analyzed program fragments. However, a similar approach using the edges of the edge-based control

flow graph as described above for formula construction involves less complex formulas because of the simpler representation of ϕ-instructions. The formula generation of SPRATTUS exploits this advantage of the edge-based interpretation of the program.

As LLVM IR enforces SSA form, each code section corresponding to an edge in the CFG is guaranteed to contain at most one definition for each program variable. Thus, only two sets of SMT variables are necessary for a formula describing an edge: one for the state before executing the corresponding instruction sequence and one for the state afterwards. In a non-SSA program representation, each new definition of a program variable inside a single fragment would require a fresh SMT variable to represent the value of the assigned variable after the definition. Note that the distinction between values before and after a program edge is only necessary because of variable definitions in loops. In such cases, the state before an edge that defines a variable can contain valid information about this variable.

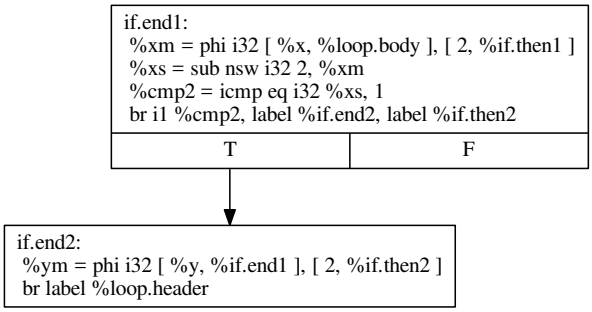


Figure 4.3: Subgraph of CFG from Figure 4.1b.

For example, for the transition $[4 \rightarrow 6]$ in Figure 4.1c that represents the non-ϕ-instructions of `if.end1` and the ϕ-nodes of `if.end2` as displayed in Figure 4.3, the constructed formula is equivalent to the following:

$$
\begin{aligned}
& (\mathit{no\_overflow} \rightarrow xs' = 2 - xm) \\
\wedge\ & (\neg \mathit{no\_overflow} \rightarrow \mathit{undef}) \\
\wedge\ & (xs' = 1 \leftrightarrow cmp2' = 1) \\
\wedge\ & (cmp2' = 0 \rightarrow \mathsf{false}) \\
\wedge\ & (ym' = y) \\
\wedge\ & (x' = x) \wedge (y' = y) \wedge (z' = z) \\
\wedge\ & (unknown' = unknown) \wedge (xm' = xm) \\
\wedge\ & (cmp1' = cmp1) \wedge (tobool' = tobool)
\end{aligned}
$$

The first four conjuncts of the formula capture the semantics of the non-ϕ-instructions of `if.end1`, with the first two conjuncts for the subtraction, the third for the comparison operation and the last one for the conditional branch instruction. By implying false if the branch that contributes to the analyzed edge is not taken, only program executions that really contribute to the edge are considered for the analysis. The ϕ-nodes from

`if.end2` is captured as a simple copy by the conjunct $(ym' = y)$ and the program variables that are not defined inside the program fragment are defined to keep their values with the remaining conjuncts.

In the formula, *no_overflow* represents a subexpression that is `true` if and only if the subtraction of the signed interpreted 32-bit numbers 2 and xm is representable as a signed interpreted bit vector of width 32. *undef* is an additional SMT variable that does not correspond to any program variable but simply indicates the use of undefined behavior in the analyzed fragment. This construction guarantees that in case of possible undefined behavior in the execution of an instruction, the computation can have any result. Although this is necessary for program verification purposes, for applications for which it is sound to assume that no undefined behavior occurs, e.g. for program transformations in a compiler, it would be more desirable to encode this assumption in the SMT formula. This can be achieved by adding a further conjunction with the expression $\neg$*undef*.

In SPRATTUS, the relation between SSA-variables and SMT variables is captured in `ValueMapping`s that specify for each SSA-variable `x` for a certain program point `P` in a certain fragment F, which SMT variable represents its current state. If `x` is defined inside F before `P`, the corresponding output SMT variable for `x` has to be used whereas in other cases, i.e. if `x` has no definition inside F or if the definition of `x` inside F occurs after `P`, the corresponding element of the set of input variables has to be chosen.

4.2.2 Fragment-Level Construction

The guarantees of the SSA form extend even further since every acyclic subgraph of an SSA control flow graph fulfills the same requirement: during execution of the instructions that comprise the subgraph, each variable is defined at most once. So, the framework for obtaining formulas for control flow edges can be extended to arbitrary acyclic fragments without modifying the `ValueMapping` mechanism.

SPRATTUS supports formula generation for subgraphs of the CFG that can be described by a single starting node S and a single ending node E. The fragment contains all edges that are a part of paths from S to E without any redundant nodes. S and E do not have to be distinct nodes of the CFG, however, if they are equal, only the non-trivial paths from S to E that contain at least one edge are considered as valid executions for this fragment. Note that even in this case no definition can be executed twice in a fragment as only the non-ϕ-instructions of the starting node and the ϕ-nodes of the ending node are contained in the fragment.

Within this construction, SPRATTUS translates the block-based CFG that is implied by the LLVM IR into SMT formulas that reflect an edge-based view on the control flow graph.

It is necessary to encode the flow of control inside a fragment if more than one edge is represented. The formulas used in SPRATTUS capture control flow by introducing a unique boolean SMT variable for each control flow edge. A model for such a formula assigns an edge variable to be true if and only if it represents the state after a computation that passes through the corresponding edge of the control flow graph.

This behavior is achieved by a general algorithm for creating the formulas for an arbitrary loop-free fragment F:

1. As before, for each edge $[a \to b]$ in F, corresponding formulas $\psi^{\bullet}_{a,exec}$ and $\psi^{\circ}_{b,exec}$ for the non-ϕ-instructions of the source block and for the ϕ-nodes of the destination block, respectively, are generated.

2. For each such $\psi^{\bullet}_{a,exec}$ and $\psi^{\circ}_{b,exec}$, formulas $\psi^{\bullet}_{a,preserve}$ and $\psi^{\circ}_{b,preserve}$ are constructed to encode that the values of all variables that are defined in the non-ϕ-part of a or in the ϕ-part of b, respectively, are not modified.

3. For each basic block a that is part of an edge of F, the condition $\psi^{\circ}_{a,cond}$ for the execution of the ϕ-part of a is computed as the disjunction over all SMT variables corresponding to edges in F with destination a. Similarly, the condition $\psi^{\bullet}_{a,cond}$ for the execution of the non-ϕ-part of a is constructed considering all edges that start in a.

4. From the previously defined subformulas, the formula ψ_{insts} is built as

$$\bigwedge_{a \in BB} \big((\psi^{\circ}_{a,cond} \to \psi^{\circ}_{a,exec}) \wedge (\neg\psi^{\circ}_{a,cond} \to \psi^{\circ}_{a,preserve}) \\ \wedge (\psi^{\bullet}_{a,cond} \to \psi^{\bullet}_{a,exec}) \wedge (\neg\psi^{\bullet}_{a,cond} \to \psi^{\bullet}_{a,preserve}) \big)$$

   where BB is the set of all basic blocks that are source or destination of an edge of F. This formula guarantees that whenever a model for it has an edge variable set, the corresponding instructions have to be taken into account and whenever an edge variable is not set, that the SSA-variables that are defined in the corresponding instructions are not modified.

5. The values of function arguments and program variables that are not defined inside F are preserved throughout the fragment by the subformula ψ_{unmod}. It contains for each such variable or argument `v` a conjunct $(v' = v)$.

6. The validity of executions with respect to control flow edges in a model is ensured by a formula ψ_{CFG} that is a conjunction of implications. For each basic block a that occurs in F and is not its starting basic block, it contains the subformula

$$\psi^{\circ}_{a,cond} \to \psi^{\bullet}_{a,cond}$$

   to enforce executions to end nowhere but in the ending point of the fragment. Similarly, it also contains formulas

$$\psi^{\bullet}_{b,cond} \to \psi^{\circ}_{b,cond}$$

for each basic block b that occurs in F except for the ending block to forbid executions that start "out of thin air". Additionally, conjuncts for each basic block stating that at most one edge to a successor block can be taken in an execution have to be added. For a basic block a with a set of successor blocks S, the necessary formula is:

$$\bigwedge_{b \in S} \left(\mathit{from\_a\_to\_b} \rightarrow \bigwedge_{c \in S \setminus \{b\}} \neg \mathit{from\_a\_to\_c} \right)$$

where $\mathit{from\_a\_to\_b}$ is the SMT variable that represents the edge from basic block a to basic block b.

7. Executions beginning in the starting point of F are allowed by the formula ψ_{start} which only contains the condition $\psi^{\bullet}_{start,cond}$ for the execution of the non-ϕ-part of the starting basic block.

The subformula construction for most instructions is completely unaware of the control flow encoding except for a few special cases:

- ϕ-nodes depend in their evaluation on the preceding block in the execution. Thus, the semantics of a ϕ-node in basic block `C` of the form

  `v = phi <ty> [`x`, A], [`y`, B]`

  that assigns to `v` the value x if execution comes from basic block `A` and y if execution comes from basic block `B` can be captured by the formula

  $$(\mathit{from\_A\_to\_C} \rightarrow v' = x) \wedge (\mathit{from\_B\_to\_C} \rightarrow v' = y).$$

- conditional branch instructions (and similarly switch instructions) constrain the executed edges, hence they should restrict the values of corresponding edge variables: A conditional branch instruction in basic block `C` of the form

  `br i1` c `A, B`

  that continues execution in basic block `B` if c is 0 and in basic block `A` otherwise can be represented by the formula

  $$(c \neq 0 \rightarrow \mathit{from\_C\_to\_A}) \wedge (c = 0 \rightarrow \mathit{from\_C\_to\_B}).$$

The resulting formula that captures the semantics of the fragment is the conjunction of the previously defined formulas:

$$\psi \equiv \psi_{insts} \wedge \psi_{unmod} \wedge \psi_{CFG} \wedge \psi_{start}.$$

Consider the function displayed in Figure 4.4 as an example for the formula generation. It returns the absolute value of the difference of its two unsigned interpreted bit vector arguments `a` and `b` by comparing them to each other and then subtracting the smaller from the greater value. As the function is acyclic, a formula describing its semantics can be constructed following the above algorithm.

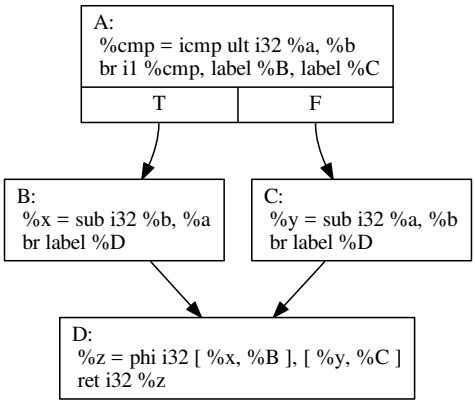


Figure 4.4: Example CFG for control flow encoding.

The subformula ψ_{insts} that captures the semantics of the instructions in the involved basic blocks can be simplified to the following formula (omitting implications of the form $x \rightarrow \mathsf{true}$):

$$
\begin{aligned}
\psi_{insts} \equiv\ & \big((\mathit{from\_A\_to\_B} \vee \mathit{from\_A\_to\_C}) \rightarrow\ ((a < b \leftrightarrow cmp' = 1) \\
& \qquad \wedge (cmp' \neq 0 \rightarrow \mathit{from\_A\_to\_B}) \\
& \qquad \wedge (cmp' = 0 \rightarrow \mathit{from\_A\_to\_C}))\big) \\
\wedge\ & (\neg(\mathit{from\_A\_to\_B} \vee \mathit{from\_A\_to\_C}) \rightarrow cmp' = cmp)) \\
\wedge\ & (\mathit{from\_B\_to\_D} \rightarrow x' = b - a) \\
\wedge\ & (\neg \mathit{from\_B\_to\_D} \rightarrow x' = x) \\
\wedge\ & (\mathit{from\_C\_to\_D} \rightarrow (y' = a - b)) \\
\wedge\ & (\neg \mathit{from\_C\_to\_D} \rightarrow y' = y) \\
\wedge\ & ((\mathit{from\_B\_to\_D} \vee \mathit{from\_C\_to\_D}) \rightarrow ((\mathit{from\_B\_to\_D} \rightarrow z' = x') \\
& \qquad \wedge (\mathit{from\_C\_to\_D} \rightarrow z' = y'))) \\
\wedge\ & (\neg(\mathit{from\_B\_to\_D} \vee \mathit{from\_C\_to\_D}) \rightarrow z' = z)
\end{aligned}
$$

In this formula, each of the four pairs of consecutive conjuncts describes the semantics of instructions from one of the basic blocks `A`, `B`, `C` and `D`, respectively.

The only SSA-variables in the fragment that are not defined in the function are the function arguments `a` and `b`, hence they are preserved with the fragment preservation formula ψ_{unmod}:

$$\psi_{unmod} \equiv a' = a \wedge b' = b$$

A valid control flow is ensured by the formula ψ_{CFG} as defined below.

$$\begin{aligned}\psi_{CFG} \equiv &(\mathit{from\_A\_to\_B} \rightarrow \mathit{from\_B\_to\_D})\\ &\wedge(\mathit{from\_B\_to\_D} \rightarrow \mathit{from\_A\_to\_B})\\ &\wedge(\mathit{from\_A\_to\_C} \rightarrow \mathit{from\_C\_to\_D})\\ &\wedge(\mathit{from\_C\_to\_D} \rightarrow \mathit{from\_A\_to\_C})\\ &\wedge((\mathit{from\_B\_to\_D} \vee \mathit{from\_C\_to\_D}) \rightarrow \mathit{from\_D\_to\_EXIT})\\ &\wedge(\mathit{from\_D\_to\_EXIT} \rightarrow (\mathit{from\_B\_to\_D} \vee \mathit{from\_C\_to\_D}))\\ &\wedge(\mathit{from\_A\_to\_B} \rightarrow \neg \mathit{from\_A\_to\_C})\\ &\wedge(\mathit{from\_A\_to\_C} \rightarrow \neg \mathit{from\_A\_to\_B})\end{aligned}$$

The first four conjuncts of the formula guarantee that control flow neither ends spontaneously nor starts in the nodes `B` and `C`. The next two subformulas assert the same for control flow at `D`, note the artificial exit node `EXIT` that is needed for non-ϕ-instructions of function ending blocks. The last two conjuncts forbid executions that take more than one of the edges [`A` → `B`] and [`A` → `C`].

Finally, the formula ψ_{start} that guarantees that one of the edges [`A` → `B`] and [`A` → `C`] has to be taken in a valid execution of the fragment is defined as:

$$\psi_{start} \equiv \mathit{from\_A\_to\_B} \vee \mathit{from\_A\_to\_C}$$

An extended formula generation for program fragments that contain loops is not implemented in Sprattus. Such fragments would lack an important property for an elegant formula construction: An execution of a loop can (re-)define the values of program variables arbitrarily often. Hence, they would require a more complex SMT variable management that would yield relatively few benefits: a formula can only represent a fixed number of loop iterations and since the precise number of iterations is often not known (and can even be unbounded), support for fragments containing loops would not remove the necessity of abstraction points inside loops.

4.2.3 Fragment Decompositions

Sprattus provides various default *fragment decompositions* that partition the input program into acyclic, connected fragments by specifying abstraction points that mark fragment starting and ending nodes. The currently supported strategies are **Edges**, **Headers**, **Bodies** and **Backedges**. They are described in the following:

Edges The most fine-grained supported fragment decomposition specifies every node in the control flow graph (i.e. every basic block) to be an abstraction point. This way, every fragment only contains the non-ϕ-instructions of the starting basic block and the ϕ-nodes of the ending basic block of a single control flow edge.

Headers A potential candidate for placing as few abstraction points as necessary is placing them at loop headers, i.e. at the entry nodes of every loop. An example of this decomposition strategy applied to the example from Figure 4.1 is shown in Figure 4.5.

Bodies Instead of inserting abstraction points at loop headers, another approach is to abstract at the starting nodes of each loop back edge in the program, i.e. at the end of every loop body. A sample decomposition using this strategy is also depicted in Figure 4.5.

Loop body fragment decompositions contain larger, overlapping fragments than similar decompositions following the header-based strategy. Hence, stronger invariants may be available at the loop condition which may yield more precise results. Especially with abstraction points before the end of the loop, the analysis can use results of computations that are associated to the back edge of a loop when analyzing the fragments that leave the loop.

Backedges The combination of both previous approaches assumes abstraction points both at loop headers and at back edge origins. This leads to smaller fragments which cause less precise information than any of the header and body strategies.

Notice that for loop-free functions, all of the mentioned strategies except for the simple edgewise decomposition generate a single fragment.

4.3 Analyzer

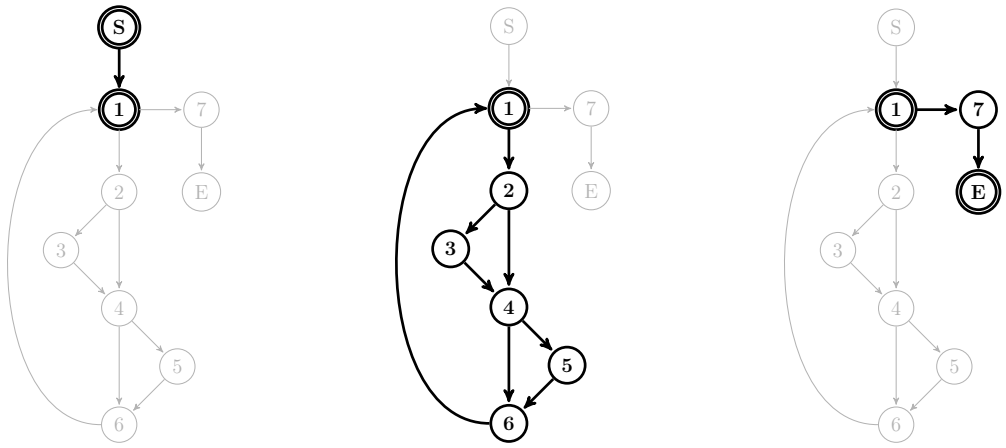
The integrated fixed-point engine of Sprattus intra-procedurally calculates analysis results for an LLVM function by computing the least fixed point of its abstract semantics (cf. [Cousot and Cousot, 1977]) with the best abstract transformers provided by symbolic abstract interpretation. The transformers are only calculated for the abstraction points given by the chosen fragment decomposition. All of the abstract domains that are associated with the abstraction points are provided by a general abstract domain interface and can be different for every program location.

The modular design of the analyzer allows to parameterize the analysis in different aspects such as the algorithms used for symbolic abstraction and the computation of the best abstract transformer. The initial set of available analyzer configurations started with implementations of the incremental unilateral algorithm presented in section 2.3.1 and an even more incrementalized version of it that exploits internal features of the SMT solver by reusing an individual solver instance for each computation of an abstract transformer for a specific fragment. For this thesis, it has been supplemented by an implementation of the bilateral algorithm from section 2.3.1.

By default, Sprattus uses a recursive, lazy algorithm presented in [Seidl et al., 2012, chapter 1.12] to obtain the least fixed point of the implied system of equalities.

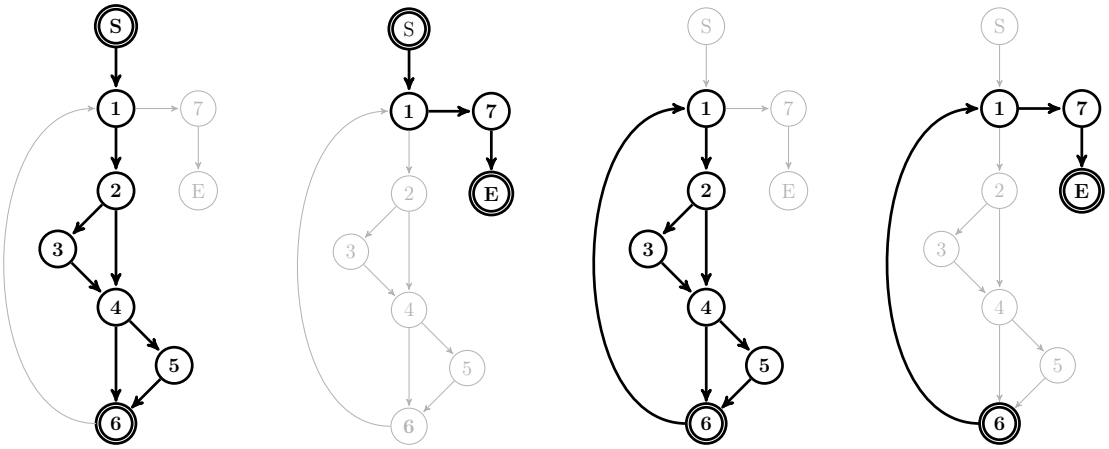
As the internal program representation relates basic blocks with the program states after the execution of all ϕ-nodes of the block and before the execution of the remaining

Strategy: `headers`



(a) (b) (c)

Strategy: `body`



(d) (e) (f) (g)

Figure 4.5: Example fragment decompositions with abstraction points at loop headers (a-c) and at the ends of loop bodies (d-g) for the program from Figure 4.1. Double circles denote fragment starting and ending points. For (b) and (f), starting and ending nodes coincide.

instructions, complete information about the state after the execution of the basic block cannot immediately be obtained from the analysis results. Nevertheless, this information is relevant for some applications, e.g. to perform the necessary analysis for a compiler optimization that argues about abstract values inside basic blocks. Therefore, the analyzer supports additional queries for such states after a basic block. To generate these states for a basic block `B`, the usual abstract values for all abstraction points from which `B` is reachable without passing other abstraction points are calculated first. Then, special abstract transformers for the subfragments starting in these preceding abstraction points and ending after all of the instructions of `B` are constructed and used to calculate the join of all corresponding abstract values. The result is the most precise information for the state after the execution of `B` that can be deduced from the preceding abstract states.

As the SMT-solving problem is NEXPTIME-complete in general (cf. [Kovásznai et al., 2012]), solver calls tend to dominate the overall run time of the analyzer by far. Thus, any worst-case complexity consideration of the analyzer can only yield run times that are at least exponential in the program size. However, the actual execution time of the solver calls may vary dramatically. More meaningful run time estimates can be achieved by calculating the number of calls to the solver that are issued at most during the analysis of a function $\mathcal{F}$ with the abstract domain $\mathcal{A}$.

Solver queries are only issued in the symbolic abstraction algorithms used for the computation of the best abstract transformers. Of the presented algorithms, both the unilateral and the bilateral algorithm perform at most $\mathit{height}(\mathcal{A})$ solver calls.

With this information, estimations for the number of abstract transformer evaluations of the used fixed-point algorithms can be used as upper bounds on the number of issued solver calls. Assuming that the used decomposition strategy divides $\mathcal{F}$ into N fragments, an upper bound for the number of performed abstract transformer evaluations in the recursive algorithm that is implemented in Sprattus is given by $\mathit{height}(\mathcal{A}) \cdot N$ in [Seidl et al., 2012]. Hence, Sprattus performs not more than $\mathit{height}(\mathcal{A})^2 \cdot N$ calls to the SMT solver during the analysis of $\mathcal{F}$.

4.4 Abstract Domains

In Sprattus, abstract domains are realized by implementing the `AbstractValue` interface. It contains the typical lattice operations like join $\sqcup$ and meet $\sqcap$, checks for equality to $\top$ or $\bot$ (cf. section 2.1) and symbolic-abstraction-specific operations such as the symbolic concretization function $\widehat{\gamma}$ and the abstract consequence operation needed for the bilateral algorithm (cf. section 2.3).

Most of the domains that have been designed before and during this thesis focus on rather limited parts of the program semantics (for example statical determined values of a single variable instead of common constructions with mappings from variables to static knowledge about their values as e.g. in [Wegman and Zadeck, 1991]). These simple elements can then be combined using a product domain as described in

subsection 4.4.6 to more expressive domains. The behavior of the above described common variable mapping construction can thus be achieved by creating a product that contains an abstract value for each variable that should be tracked. The framework provides `Product::ForValues()` and `Product::ForValuePairs()` functions to create such products for all reachable SSA-variables in a given basic block.

As many of the so far implemented domains have a small height, for many of them no better abstract consequence operation than returning the first argument can be implemented. If not stated otherwise, the described abstract domains only support the always sound default abstract consequence operation that returns the first argument without modification.

4.4.1 Constant Propagation

The *Constant Propagation* domain describes for an SSA-variable `x` whether it has the same constant value in every feasible program run and if so, which value that is. It consists of a flat lattice of the possible variable values (Figure 4.6).

A constant value in the domain signals that the respective variable has this value in every program run, $\top$ denotes that the constant could not be proven to have the same value in every program run, and $\bot$ denotes that the variable has no value in any program run because it is unreachable.

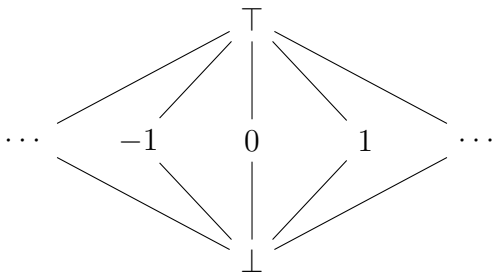


Figure 4.6: *Constant Propagation* lattice.

The symbolic concretization function $\widehat{\gamma}_{CP}[\mathtt{x}]$ and the representation function $\beta_{CP}[\mathtt{x}]$ for an instantiation of this domain that tracks the value of the SSA-variable `x` are straightforwardly defined (where $[\ldots, \mathtt{x} = v, \ldots]$ is a concrete state that contains a mapping from `x` to the value v):

$$\widehat{\gamma}_{CP}[\mathtt{x}](A) = \begin{cases} \mathsf{false} & : A = \bot \\ \mathsf{true} & : A = \top \\ \mathtt{x} = A & : \text{otherwise} \end{cases}$$

$$\beta_{CP}[\mathtt{x}]([\ldots, \mathtt{x} = v, \ldots]) = v$$

The representation function $\beta_{CP}[\mathtt{x}]$ maps a concrete state just to the value that it assigns to `x`. It is impossible to get an overdefined or underdefined value for representing

a single concrete state, as the concrete state has to provide exactly one assignment for each variable. Notice that in the implementation, concrete states are based on models of the SMT solver. Therefore, even for transformers that do not define any mapping for a variable `x`, the SMT solver completes concrete states to contain a mapping for `x` to some arbitrary, solver-dependent value.

The domain is implemented in a way such that it supports SSA-variables of any type that is representable in the SMT semantics. In the current implementation these are bitvectors and floating point variables. As a generic flat lattice, its height is 3.

Because of the structure of LLVM IR, which enforces condition variables for every conditional branch, this domain can also be used to track control flow and basic block reachability: The abstract value corresponding to a branch condition will be `1` if and only if the `false`-Successor of the parent basic block can be proven unreachable from this basic block by Sprattus and `0` if and only if the `true`-Successor is proven unreachable from the parent basic block by Sprattus.

This way, the *Constant Propagation* domain of Sprattus alone is sufficient for the analyses necessary for the well known *Sparse Conditional Constant Propagation* transformation [Wegman and Zadeck, 1991].

4.4.2 Scalar Relations

For some applications, abstract states only describing properties concerning single variables are not expressive enough to represent the desired results. In this case, considering relations between multiple program variables can improve the expressiveness of an analysis significantly. One example for such a relational domain is the *Scalar Relations* domain implemented in Sprattus. It tracks the relation of two scalar SSA-variables `x` and `y` in a program. Possible relations are strict scalar comparisons and equality as well as all the combinations thereof. Hence, the corresponding lattice can be described as the power set lattice of the set $\{<, =, >\}$ where non-singleton sets represent the disjunction of the contained elements. For example, the abstract state $\{<, =\}$ soundly over-approximates every concrete state in which the value of `x` is either less than or equal to the value of `y`. Thus, this state can also be described as $\leqslant$, as done in Figure 4.7. The other subsets of size two, $\{=, >\}$ and $\{<, >\}$, are similarly described by the respective combined relations $\geqslant$ and $\neq$. $\top$ represents that the values of `x` and `y` could be in any relation whereas $\bot$ represents concrete states in which `x` and `y` are in no relation, i.e. no states at all.

As a power set lattice over a set containing three elements, the height of the *Scalar Relations* lattice is 4.

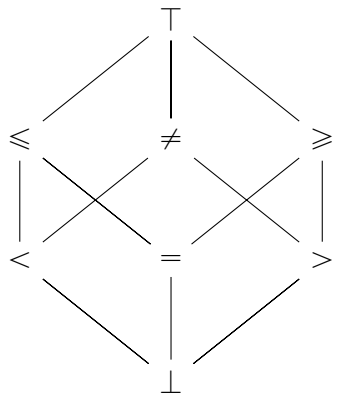


Figure 4.7: *Scalar Relations* lattice.

Conforming with their intuitive meaning as described above, symbolic concretization and representation functions for this domain are defined as follows:

$$\widehat{\gamma}_{SR}[\mathtt{x},\mathtt{y}](A) = \begin{cases} \mathsf{false} & : A = \bot \\ \mathsf{true} & : A = \top \\ \mathtt{x}\,\mathrm{A}\,\mathtt{y} & : \text{otherwise} \end{cases}$$

$$\beta_{SR}[\mathtt{x},\mathtt{y}]([\ldots,\mathtt{x}=v,\mathtt{y}=w,\ldots]) = \begin{cases} < & : v < w \\ = & : v = w \\ > & : v > w \end{cases}$$

The *Scalar Relations* domain can be of particular use in the context of branch conditions depending on the relation between two program variables. Here, non-trivially unreachable code sections can be entirely eliminated with the help of an analysis incorporating this domain.

4.4.3 Predicates

The *Predicates* domain tracks the validity of an arbitrary binary predicate ψ over two SSA-variables `x` and `y`.

Its behavior can be described by a generic flat lattice over the set $\{\mathtt{true},\mathtt{false}\}$ (Figure 4.8). Therein, `true` and `false` represent the fact that the predicate for the chosen SSA-variables is always true or always false, respectively. $\top$ and $\bot$ stand for the predicate not being proven to always have the same truth value or not having any truth value, respectively.

If the second variable for the construction of a *Predicates* domain is omitted, occurrences of the second variable in the predicate are replaced by zero values of the type of the first variable. This allows using a single predicate both for relational and non-relational analyses. The height of this domain is 3 since it is a flat lattice.

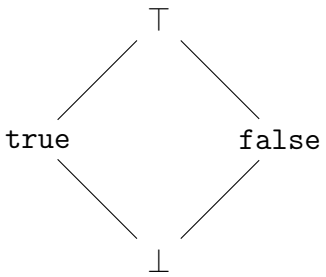


Figure 4.8: *Predicates* lattice.

These design choices reflect directly in the definitions of $\widehat{\gamma}_{BP}[\psi, \mathtt{x}, \mathtt{y}]$ and $\beta_{BP}[\psi, \mathtt{x}, \mathtt{y}]$ for abstract values covering the validity of the binary predicate ψ for the variables $\mathtt{x}$ and $\mathtt{y}$ as well as the derived non-relational versions $\widehat{\gamma}_{BP}[\psi, \mathtt{x}]$ and $\beta_{BP}[\psi, \mathtt{x}]$.

$$\widehat{\gamma}_{BP}[\psi, \mathtt{x}, \mathtt{y}](A) = \begin{cases} \mathsf{false} & : A = \bot \\ \mathsf{true} & : A = \top \\ \psi[\mathtt{x}, \mathtt{y}] & : A = \mathtt{true} \\ \neg\psi[\mathtt{x}, \mathtt{y}] & : A = \mathtt{false} \end{cases}$$

$$\beta_{BP}[\psi, \mathtt{x}, \mathtt{y}]([\ldots, \mathtt{x} = v, \mathtt{y} = w, \ldots]) = \begin{cases} \mathtt{true} & : \psi[v, w] \\ \mathtt{false} & : \text{otherwise} \end{cases}$$

$$\widehat{\gamma}_{BP}[\psi, \mathtt{x}](A) = \begin{cases} \mathsf{false} & : A = \bot \\ \mathsf{true} & : A = \top \\ \psi[\mathtt{x}, 0] & : A = \mathtt{true} \\ \neg\psi[\mathtt{x}, 0] & : A = \mathtt{false} \end{cases}$$

$$\beta_{BP}[\psi, \mathtt{x}]([\ldots, \mathtt{x} = v, \ldots]) = \begin{cases} \mathtt{true} & : \psi[v, 0] \\ \mathtt{false} & : \text{otherwise} \end{cases}$$

In the following, some instantiations of the *Predicates* domain that have been implemented for this thesis are presented.

Equality

A very simple but expressive relational domain that is implementable as a *Predicates* domain is the *equality* domain. The predicate is

$$\psi(a, b) := (a = b)$$

which tracks the equality of two SSA-variables that are represented in Z3 with the same type.

This domain is strictly less expressive than the Scalar Relations domain (subsection 4.4.2) as it only distinguishes between equality and inequality. Hence, an analysis using the Scalar Relations domain will yield at least as precise results as one using the *Equality* Predicate domain. However, the *Equality* Predicate domain benefits from the reduced lattice height which directly influences the overall run time.

The information gathered with this domain can be used to create a global value numbering [Kildall, 1973], as an aggressive form of a copy propagation analysis and as a simple intraprocedural alias analysis.

For the purpose of an immediate use for global value numbering or copy propagation, it can be sufficient to only track equalities between SSA-variables of which one could possibly be replaced by another in a certain basic block `bb`. This is the case if one of the variables is used in an instruction `i` in `bb` and the definition of the other one dominates `i`.

For this thesis, a variation of the previously described `Product::For*()` functions has been implemented to create a reduced product of abstract values for equality tracking for these restricted variable pairs.

Equality to `null`

The non-relational variant of the *Equality* domain tracks equality and inequality to an appropriate `null` value for a single SSA-variable. The predicate is thus reduced to:

$$\psi(a) := (a = 0).$$

Simple equality to 0 can also be expressed in the *Constant Propagation* domain, the power of this domain lies in the representation of definite inequality to 0. Expressing this relation can be useful in the context of C-like condition evaluation, i.e. `null` corresponds to a false value, everything else is a true value. Checks for `null` pointers and division-by-zero errors can also be proven obsolete with this domain.

Parity

A rather specialized instantiation of the *Predicates* domain is the *Parity* domain. This non-relational domain uses the predicate

$$\psi(a) := (a \bmod 2 = 0)$$

to check for a program variable of bit vector type whether it is even or odd if interpreted as unsigned integer. A `true` abstract value thus represents a variable with always even values whereas a `false` value represents a variable that only contains odd values during any computation.

4.4.4 BitMasks

For creating a domain, it is not necessary to interpret bit vectors as signed or unsigned integers. For example, the *BitMasks* domain ignores any interpretation of the bit vec-

tors: it tracks must-information on the values of the separate bits of the values that an SSA-variable can have in an execution.

This abstract domain can be formalized for a given bit width $\texttt{bw}$ as $\mathcal{BM}_\bot = (\{0, \ldots, \texttt{bw}-1\} \to \{\texttt{0}, \texttt{1}, *\}) \cup \{\bot\}$, the set of functions that map positions in a bit vector to either a definite zero, definite one or a statically unknown value, extended with a separate bottom element $\bot$. The $\top$ element of this domain is the function $(\lambda x \,.\, *)$ that represents no information about any position of the bit vector. From the information of two states a and b, their join can be obtained as

$$a \sqcup b = \begin{cases} a & : b = \bot \\ b & : a = \bot \\ \lambda n \,.\, \begin{cases} c & : c = a(n) = b(n) \\ * & : \text{otherwise} \end{cases} & : \text{otherwise} \end{cases}$$

A maximal ascending chain in the lattice with the order that is induced by the definition of $\sqcup$ contains apart from $\bot$ and $\top$ for each natural number k that is less than the bit width exactly one abstract state that contains k assignments to $*$. Hence, the height of the lattice is $\texttt{bw} + 2$.

A notion of the appropriate symbolic concretization function and the representation function for this domain can be introduced as follows:

$$\widehat{\gamma}_{BM}[\texttt{x}](a) = \begin{cases} \mathsf{false} & : a = \bot \\ \bigwedge_{i \in \{0,\ldots,\texttt{bw}-1\} \wedge a(i) \neq *} \texttt{x}[i] = a(i) & : \text{otherwise} \end{cases}$$
$$\beta_{BM}[\texttt{x}]([\ldots, \texttt{x} = v, \ldots]) = \lambda n \,.\, v[n]$$

However, in the implementation, this domain is realized using bitwise operations on bitvectors to generate more compact formulas.

The *BitMasks* domain can be useful in the context of analyzing complex expressions containing bitwise operations. For implementations that encode information in the particular bits of values, it performs as a more fine grained version of the *Constant Propagation* domain.

4.4.5 Intervals

A Sprattus counterpart for the well-known interval analysis as described e.g. in [Nielson et al., 1999, example 4.10] has also been implemented for this thesis. It determines two values for an SSA-variable that denote upper and lower bounds for the values that the SSA-variable may contain in an execution. For the corresponding lattice (Figure 4.9), the intervals are ordered by set inclusion and an additional $\bot$ element is introduced. Bit vector values are interpreted as signed integers for this domain.

As the analysis is designed for LLVM IR, the possible values for these bounds have to lie within the range of the SSA-variable's type, therefore no additional $\top$ element

is needed. The unique maximal element of the lattice is $[\texttt{MIN}, \texttt{MAX}]$ where `MIN` and `MAX` are the minimal and maximal representable values for the variable type. A maximal ascending chain for this lattice starts from $\bot$ and contains for each possible interval cardinality an interval of this cardinality. There is an interval for each cardinality from 1 to $2^{\texttt{bw}}$ where `bw` is the bit width of the tracked variable, hence the height of the lattice is $2^{\texttt{bw}} + 1$.

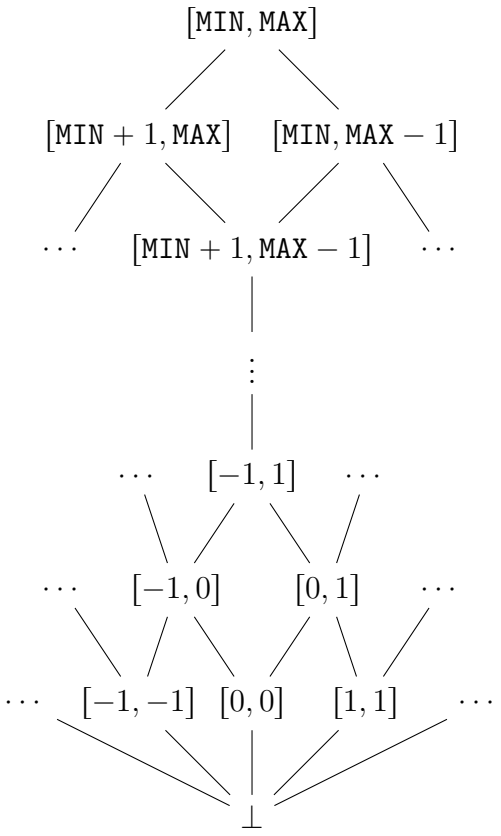


Figure 4.9: *Intervals* lattice.

Another consequence of the limited value range is that — in contrast to typical theoretical interpretations of the *Intervals* domain — the domain for this lattice has finite height. Therefore, no widening [Cousot and Cousot, 1977] is needed to ensure termination of fixed point algorithms. Nevertheless, the immense lattice height of $2^{\texttt{bw}}+1$ yields enough motivation for simplified versions of the *Intervals* domain.

An example for such a simplification of the *Intervals* domain that has been implemented for this thesis is the domain of *Threshold Intervals* that has a limited set of threshold values which are considered for either upper or lower bounds of non-singleton intervals. Singleton intervals are still represented precisely. By specifying the thresholds, the analysis designer can choose any trade-off between complexity and expressiveness of the domain: the lattice height is defined by $\#\texttt{thresholds} + 1$. Possible sets of thresholds would be e.g. the set of powers of two that lie within the variable's ranges or the set of constants occurring in the program source code.

Both the *Threshold Intervals* and the standard *Intervals* domain rely on the following straightforward symbolic concretization and representation functions:

$$\widehat{\gamma}_I[\mathtt{x}](A) = \begin{cases} \mathsf{false} & : A = \bot \\ \mathtt{x} \geqslant a \wedge \mathtt{x} \leqslant b & : A = [a, b] \end{cases}$$
$$\beta_I[\mathtt{x}]([\ldots, \mathtt{x} = v, \ldots]) = [v, v]$$

The *Intervals* domain is a lattice of sufficient height to provide a reasonable non-trivial abstract consequence operation. For two abstract states $[l_1, u_1]$, $[l_2, u_2]$ from the *Intervals* domain such that $[l_1, u_1] \sqsubsetneq [l_2, u_2]$ (i.e. $l_2 \leqslant l_1 \wedge u_1 \leqslant u_2$ and $l_1 \neq l_2 \vee u_1 \neq u_2$), `abstractConsequence`$([l_1, u_1], [l_2, u_2])$ returns the abstract value

$$\left[l_1 - \left\lfloor \frac{l_1 - l_2}{2} \right\rfloor, u_1 + \left\lfloor \frac{u_2 - u_1}{2} \right\rfloor\right]$$

This produces an abstract consequence with bounds exactly in the middle of the respective bounds of the arguments as depicted in Figure 4.10. Thus, the bilateral algorithm performs a binary search on both interval bounds.

Figure 4.10: Abstract consequence operation for *Intervals*, with $[l, u]$ = `abstractConsequence`$([l_1, u_1], [l_2, u_2])$.

4.4.6 Reduced Product

The key to abstract domains of significant expressiveness lies in combining simple domains into more complex ones. In Sprattus, this is done by composing the simpler abstract domains in a *Reduced Product* abstract domain. It is implemented using an extension to arbitrary numbers of component domains of the technique described in subsection 2.3.2 and hence needs no additional modification of the component domains.

The symbolic concretization function as well as the representative function of the product generalize the respective functions given in subsection 2.3.2 to k component domains:

$$\widehat{\gamma}_P[\mathcal{A}_1[p_1], \ldots, \mathcal{A}_k[p_k]]((a_1, \ldots, a_k)) = \bigwedge_{i=1}^{k} \widehat{\gamma}_{\mathcal{A}_i}[p_i](a_i)$$
$$\beta_P[\mathcal{A}_1[p_1], \ldots, \mathcal{A}_k[p_k]](\sigma) = (\beta_{\mathcal{A}_1}[p_1](\sigma), \ldots, \beta_{\mathcal{A}_k}[p_k](\sigma))$$

As this abstract domain describes the reduced product of k domains, its height depends not only on the height of the component domains but also on the amount of

information that can be represented in both and would thus be redundantly represented in a direct product. For worst-case considerations, one has to expect the represented information in the component domains to be completely disjoint and hence forming a direct product with height $\sum_{i=1}^{k} height(\mathcal{A}_i[p_i])$. However, significantly less states in an actual maximal ascending chain are possible, a tight lower bound would be the maximal height of any component domain.

The potentially large number of component values in a product abstract value enables the use of a non-trivial abstract consequence operation for this domain. For the defining requirements on an abstract consequence operation in products to hold, it is sufficient to recursively compute the abstract consequence for any strictly positive number of components of the argument abstract values in which the component of the first argument is strictly smaller than the respective component of the second argument and to set every other component to $\top$:

$$\begin{aligned}&\texttt{absCons}((l_1,\ldots,l_k),(u_1,\ldots,u_k)) = \\ &\qquad(\top,\ldots,\top,\texttt{absCons}_{i_1}(l_{i_1},u_{i_1}),\top,\ldots,\top,\texttt{absCons}_{i_p}(l_{i_p},u_{i_p}),\top,\ldots,\top) \\ &\quad\text{for } i_j \in I \text{ with } I \subseteq \{n \mid l_n \sqsubsetneq u_n\} \text{ such that } p := |I| \geqslant 1\end{aligned}$$

This abstract consequence operation significantly differs from the solution suggested in [Thakur, 2014] for conjunctive domains: In the setting of the *Reduced Product* domain, the operation proposed by Thakur is equivalent to setting every component to $\top$ except for one component from the first argument that is not equal to or higher in the lattice than the corresponding component in the second argument. Compared to the here proposed implementation, this abstract consequence operation can lead to a larger number of solver calls.

For proving the above defined function `absCons` to be a valid abstract consequence operation, two statements have to be proven:

- $\forall l \sqsubsetneq u.\ l \sqsubseteq \texttt{absCons}(l,u)$
- $\forall l \sqsubsetneq u.\ \texttt{absCons}(l,u) \not\sqsupseteq u$

Proof. The first statement follows immediately from the definition of `absCons`: each component value a_i of $\texttt{absCons}(l,u)$ is either $\top$ or the abstract consequence $\texttt{absCons}(l_i,u_i)$ of the corresponding components of l and u. By definition, $\top$ is greater than every other possible value and $\texttt{absCons}(l_i,u_i)$ has to fulfill the defining conditions of an abstract consequence operation, thus it is also greater to or equal than the corresponding component l_i of l. Hence, the inequality holds for each component of the product abstract values and therefore also for the complete product values.

For the second statement, consider the following equivalence:

$$(a_1, \dots a_k) \sqsubseteq (b_1, \dots b_k) \equiv \bigwedge_{i=1}^{k} a_i \sqsubseteq b_i$$
$$\Leftrightarrow (a_1, \dots a_k) \not\sqsubseteq (b_1, \dots b_k) \equiv \bigvee_{i=1}^{k} a_i \not\sqsubseteq b_i$$

This implies that a product abstract value b is not greater than or equal to another product abstract value a if at least one of the component abstract values of b is not greater than or equal to the corresponding component of a. As the definition of `absCons` states, at least one component c_i of its resulting abstract value $c = \texttt{absCons}(a, b)$ is an abstract consequence $\texttt{absCons}(a_i, b_i)$ of components of the arguments a and b. By the definition of the abstract consequence operation, the following holds:

$$\begin{aligned} &\texttt{absCons}(a_i, b_i) \not\sqsupseteq b_i \\ \Rightarrow &\texttt{absCons}((a_1, \dots, a_k), (b_1, \dots, b_k)) \not\sqsupseteq (b_1, \dots, b_k) \end{aligned}$$

Therefore, the defined `absCons` function is a valid abstract consequence operation. □

The relative position of the abstract consequence of a product value "between" the arguments depends on the number p of components that are not set to $\top$: the more components are set to $\top$, the higher in the lattice is the abstract consequence. Hence, it is a natural choice of design to craft the abstract consequence operation of *Product* abstract values parameterized in p as it is done in the Sprattus implementation. Here, a percentage P_{max} can be specified to configure the abstract consequence operation to prevent a randomly chosen subset of at most $\texttt{max}(1, P_{max} \cdot k)$ from being set to $\top$.

4.5 Application in Compiler Optimizations

A main contribution of this thesis is the implementation of appropriate versions of two known intraprocedural compiler optimizations on LLVM IR using the Sprattus framework. They are implemented as a highly configurable LLVM function pass that can be used at any point within the LLVM optimization pipeline.

The optimizations that have been implemented are

- a **Constant Replacement** transformation that replaces uses of SSA-variables that Sprattus can statically prove constant with their value, and
- a **Redundant Computation Elimination** transformation that replaces uses of SSA-variables with uses of other variables with equal values.

These program transformations are realized in a single optimization pass that can be configured to perform any combination of the transformations with the analysis results provided by Sprattus.

By construction of the framework, the optimization pass is parametric in all of the aforementioned options that SPRATTUS provides. This includes the symbolic abstraction algorithm to apply, the fragment decomposition to use for the best abstract transformers and the abstract domains that determine the expressiveness and precision of the analysis.

4.5.1 Constant Replacement

The Constant Replacement transformation is inspired by the Sparse Conditional Constant Propagation (SCCP) transformation proposed in [Wegman and Zadeck, 1991]. It already combines two program optimizations: constant propagation and unreachable code elimination. Of those, the former reduces run-time computations by performing as many operations with statically determined constant result as possible during compilation time. The latter removes code parts that cannot be executed in any program run and consequently reduces the size of the resulting program and eliminates unnecessary branch instructions.

The transformation uses and requires the *Constant Propagation* abstract domain presented in subsection 4.4.1 to track information about constant values of variables. Hence, the optimization has the same restriction to SSA-variables of bit vector type as the *Constant Propagation* domain. In the LLVM scenario, it is not necessary to introduce an additional domain for obtaining reachability information. LLVM branch conditions are necessarily represented as bit vector SSA-variables, so statically determined branch destinations can be analyzed by computing information on statically constant values of these variables.

However, there are conceptual differences between the SPRATTUS Constant Replacement transformation and SCCP. The SPRATTUS implementation computes for each basic block of the analyzed function whether the occuring uses of SSA-variables always yield the same constant result whereas the classical SCCP (and also the default LLVM implementation thereof) exploits the sparse program representation and only calculates for each SSA-variable in the program whether it has the same constant value in every execution of the function.

Thus, the Constant Replacement transformation as implemented in SPRATTUS is in this respect strictly more expressive than SCCP. The strictness of this relation can be observed for example when optimizing a function that contains a structure as displayed in Figure 4.11: Here, the presented transformation is able to prove that `x` is constant when it is used in any instruction from basic block `B` and can perform subsequent transformations based on this result whereas standard SCCP lacks this information and cannot state any non-trivial information about the occuring variables.
Furthermore, the implementation presented here only replaces uses of the variables that it proved to be constant by the respective constants or special `undef` values if they cannot have any value in an execution, i.e. they are never reached in execution. Especially, it does not remove any newly unused variable definitions or unreachable basic blocks. These remaining tasks can be performed by purely syntactical transformations, like removing instructions whose results are never used and replacing conditional branch

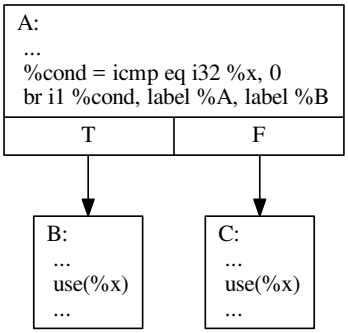


Figure 4.11: Example CFG fragment.

instructions with constant conditions by unconditional branches. For these transformations, appropriate LLVM passes such as *Dead Code Elimination* and *SimplifyCFG* exist and can be used for clean-up operations after the Sprattus pass.

It is worth noticing that the Constant Replacement transformation is more than just a syntactical (sparse conditional) constant propagation. The C program given in Figure 4.12 is a case where the Sprattus pass can provide better results than the transformations implemented in common C compilers.

```
1  int pow(int b, int e) {
2    int res = 1;
3    int i;
4    for (i = 0; i < e; i++) {
5      res *= b;
6    }
7    return res;
8  }
9
10 int foo(void) {
11   ...
12   int u = unknown();
13   int v = pow(u, 4);
14   assert(v >= 0);
15   ...
16 }
```

Figure 4.12: Example C program for Constant Replacement.

Here, the function `pow(x, y)` naively computes the power x^y of its arguments. Hence, the value of `v` is always non-negative if no overflow occurs ($x^4 \geqslant 0$ holds for each $x \in \mathbb{Z}$). Since signed integer overflows entail undefined behavior (cf. [ISO 9899:2011, 2011]), a compiler transformation can soundly remove the assertion in `foo()`.

The Clang compiler for example however fails to do so. Within its optimization pipeline, the call to `pow()` is inlined and the corresponding loop is unrolled, but it is not able to prove the condition of the assertion statically as true. The Constant Replacement transformation, if applied to the IR that is generated by `Clang` (after inlining and loop unrolling), proves the assertion condition `(v >= 0)` true for each valid execution that does not entail undefined behavior and therefore replaces it with a constant value. It is an easy task for the aforementioned clean-up passes to remove the assertion afterwards. So, the instructions for computing the conditions and those for handling the error case are removed.

4.5.2 Redundant Computation Elimination

The second program transformation that has been developed, the Redundant Computation Elimination, reduces the number of executed instructions by avoiding unnecessary recomputations of already available values. Its effects are similar to a common-subexpression elimination as achieved with the Global Value Numbering approach described in [Kildall, 1973] or the more sophisticated equality detection algorithms presented in [Alpern et al., 1988].

This program optimization is based on the results provided by the *Equality Predicates* abstract domain as described in section 4.4.3. However, not all possible combinations of SSA-variables that are represented with the same type in Z3 can be used for Redundant Computation Elimination. The definitions of potential replacement candidates for a use of an SSA-variable have to dominate the considered use and their LLVM types have to agree.

A restricted *Equality* domain that provides only this information that could possibly be used for Redundant Computation Elimination has been implemented for this thesis. It only tracks for each basic block `b` the relation between each SSA-variable `v` that is used inside `b` on one hand and each variable that has the same LLVM type as `v` and whose definition dominates a use of `v` in `b` on the other hand. From these available candidates, one has to choose a variable that is beneficial to use instead of `v` at the considered program location.

The heuristic for the profitability of replacement candidates that is implemented is based on the dominance order $\succeq$ on the set *candidates* of the definitions of the candidates[1]: Each use of `v` is replaced by a use of the maximum of all the available candidate SSA-variables that are proven to be equal (regarding the dominance order). As dominance is a partial order as well as a tree order and all the candidate definitions dominate the considered use, the candidate definitions are totally ordered by dominance:

$$(\forall x, y, z.\ (x \succeq z \wedge y \succeq z) \rightarrow x \succeq y \vee y \succeq x) \wedge (\forall r \in \mathit{candidates}.\ r \succeq \mathtt{v})$$
$$\Rightarrow \forall r, s \in \mathit{candidates}.\ r \succeq s \vee s \succeq r$$

Hence, the "maximum dominating" candidate is well-defined and contained in the set of available candidates.

Like the Constant Replacement transformation, this optimization only replaces uses. Thus, a subsequently executed clean-up pass like *Dead Code Elimination* is required to syntactically eliminate the computations that are proven to be unnecessary by Sprattus.

The LLVM project contains a GVN transformation pass that performs similar optimizations. In addition to simple redundancy of SSA-variables, it performs non-trivial analysis of memory contents for dead-load elimination which can only be realized in Sprattus by enabling an expensive memory model for the analysis. The LLVM GVN

[1] A node u in a CFG dominates a node v if and only if every path from the starting node to v contains u.

pass is also able to introduce new ϕ-nodes to reuse computations from basic blocks that might be executed before the considered use but do not dominate the use.

However, just like the standard global value numbering from [Kildall, 1973], it is syntax-based and hence not able to detect all equivalent values. For example, in [Buchwald, 2015] it has been shown that at the current state, no transformation in the `-O3` optimization pipeline of LLVM is able to prove the equivalence

$$-(x \,\&\, \texttt{0x80000000}) = x \,\&\, \texttt{0x80000000}$$

as it would occur in the following pseudo-LLVM program fragment:

```
1  %y = and i32 %x, 0x80000000
2  %z = sub i32 0x00000000, %y
3  ret i32 %z
```

The Sprattus Redundant Computation Elimination discovers that `%z` always has the same value as `%y` and is therefore able to replace the use of `%z` in line 3 by a use of `%y`. This makes the instruction in line 2 dead and easily removable by the DCE pass.

4.5.3 Improving Performance

Both of the previously described transformations can be improved by enriching the corresponding analysis with additional abstract domains. These enable Sprattus to detect more potential program executions as infeasible and therefore irrelevant for the analysis results.

Consider for example the control flow graph in Figure 4.13. It contains a function argument `%a` for which no static information is available. Nevertheless, `%x` is guaranteed to have the second-to-least significant bit set and thus cannot be equal to zero. Therefore, the comparison in `B` has to yield a `false` result and consequently, `C` is unreachable. In the optimal case, the Constant Replacement transformation would replace the use of `%z` in `E` by the constant 7 rendering all other instructions dead.

However, with its default settings (i.e. *Constant Propagation* domain and **Edges** fragment decomposition strategy), said transformation is not able to replace anything in this example: The abstract value at the node corresponding to basic block `B` only contains the information that `%a` and `%x` cannot be proven to be constant. This information is not sufficient to exclude any of the branch destinations in `B`. This issue can be overcome by adding the *Equality to* `null` abstract domain described in section 4.4.3. It provides the necessary information that `%x` is not equal to zero at `B` allowing subsequent simplifications.

The fragment decomposition is another effective factor for increasing the expressiveness of the presented program transformations. In larger fragments, no additional overapproximation is performed, resulting in fewer infeasible executions contributing to the analysis.

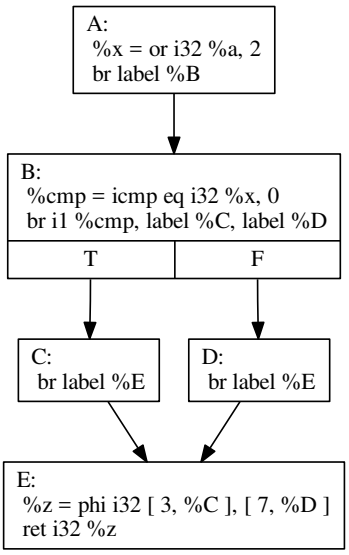


Figure 4.13: Example CFG with an infeasible path, basic block `C` is unreachable.

Returning to the example from Figure 4.13, the desired results can also be achieved by choosing a decomposition strategy that contains no abstraction point at `B` like e.g. **Bodies**. In this case, an abstract transformer for the entire function is used for the analysis without over-approximating possible executions at any point.

Sprattus is also capable of excluding executions that involve certain kinds of undefined behavior, such as division-by-zero errors from analysis considerations. This can reduce the search space for the SMT solver and consequently improve its performance. For the purpose of compiler optimizations, this is a valid approach as it is legitimate to assume correct input programs that do not contain any undefined behavior. This allows more aggressive optimizations compared to the default configuration of Sprattus that allows every kind of result for computations including undefined behavior.

Other features of Sprattus that are not considered in this thesis such as non-trivial memory models and support of floating-point arithmetics provide further expressiveness in a trade-off for increased analysis time.

5 Evaluation

For the empirical evaluation of the described contributions of this thesis, the Sprattus-based compiler optimizations have been tested on a broad set of practically relevant benchmarking suites, namely the SingleSource benchmarks of the LLVM 3.6 test suite[1]. All runs of the Sprattus pass that contribute to the presented data are performed on IR that has already run through the standard LLVM optimization pipeline in the most aggressive configuration, `-O3`. Hence, the transformations that the Sprattus pass performs are improvements compared to the set of optimizations that LLVM provides. Performance data has been measured with an Intel Core i5-4590 CPU (3.30 GHz, 6MB cache) and 12 GB RAM with a time limit of 1800 s per optimization run.

The obtained results, which are presented in the following, give answers to questions concerning the performance of the Sprattus framework and the underlying algorithms, the expressiveness of the presented program transformations and the general use of reduced products in the context of compiler optimizations.

For all of the test runs that are described in the following to use an abstract domain $\mathcal{A}$, the domain contains for each program location B a product of instantiations of $\mathcal{A}$ for all available SSA-variables at B. Consequently, if $\mathcal{A}$ is a non-relational domain, the resulting product for the location B contains an entry $\mathcal{A}[\mathtt{x}]$ for each variable $\mathtt{x}$ whose definition dominates B. If $\mathcal{A}$ is a relational domain, the product for B contains a component $\mathcal{A}[\mathtt{x}, \mathtt{y}]$ for each pair $(\mathtt{x}, \mathtt{y})$ of variables whose definitions dominate B and that have a type that is represented with the same Z3 type.

The only exception from this is the *Restricted Equality* domain that is needed for the Redundant Computation Elimination transformation. It is a relational domain whose product for B contains an instance of the *Equality Predicates* domain (cf. section 4.4.3) for each pair $(\mathtt{x}, \mathtt{y})$ of distinct SSA-variables where $\mathtt{x}$ is used in B, $\mathtt{y}$ dominates B, and both have the same LLVM type (i.e. if $\mathtt{x}$ and $\mathtt{y}$ had the same values, $\mathtt{y}$ could be used inside B to replace a use of $\mathtt{x}$).

As described in chapter 4, Sprattus is parameterized in the symbolic abstraction algorithm, the abstract domains and the fragment decomposition strategy. Benchmarking results for different parameter combinations in these aspects are presented in the following.

[1] `http://llvm.org/docs/TestingGuide.html#test-suite-overview`

5.1 Analyzer Implementations

For the analyzer, implementations of the bilateral algorithm and the standard version of the unilateral algorithm as well as the incrementalized version as described in section 4.3 are compared. The results are generated with the *Constant Propagation* abstract domain and the **Edges** fragment decomposition strategy assuming no undefined behavior.

Notice that all of the presented algorithms are guaranteed to give the same results, hence it would be of no use to compare their expressiveness in the analysis results. Therefore, only the relative analysis times of the different analyzer implementations are considered here.

5.1.1 Abstract Consequence for Products

As described in section 2.3.1, the performance of the bilateral symbolic abstraction algorithm depends heavily on the abstract consequence operation that is used. All of the presented abstract domains are implemented to express very restricted program properties that only consider single variables or pairs of variables. For an actual program analysis, abstract values of these domains are composed in larger product domains. Thus, the parameterized abstract consequence operation of the product domain (cf. subsection 4.4.6) is a promising candidate for influencing the runtime of the bilateral algorithm.

The key figure for the proposed product abstract consequence operation is the number of components for which all information is discarded. Sprattus provides an interface to specify the percentage p of the component abstract values of a product value that should not be subject to such a discarding operation.

On one hand, if p is chosen to be 0%, the abstract consequence operation yields a value that has exactly one component that is not at its $\top$ value. Hence, the resulting formulas are relatively simple, but more solver calls might be necessary to reach the final result[2].

On the other hand, for $p = 100\%$, all of the component values are chosen not to be discarded, so the resulting operation is a component-wise application of the respective abstract consequence operations. For most of the domains that are described in this thesis, these operations just return the lower bound, thus the bilateral algorithm performs the same steps as the unilateral algorithm: the lower bound is refined with models until the final result is reached, afterwards one additional call with an unsatisfiable formula is issued and the upper bound is adjusted to be equal to the lower bound.

Figure 5.1 shows an excerpt from benchmarking runs with the bilateral algorithm and different values for p. Overall, the configurations are evaluated on 126 benchmarks from the LLVM SingleSource benchmark suite. Of these, 9 of the benchmarks cause

[2] this closely resembles the abstract consequence operation for conjunctive domains as proposed in [Thakur, 2014]

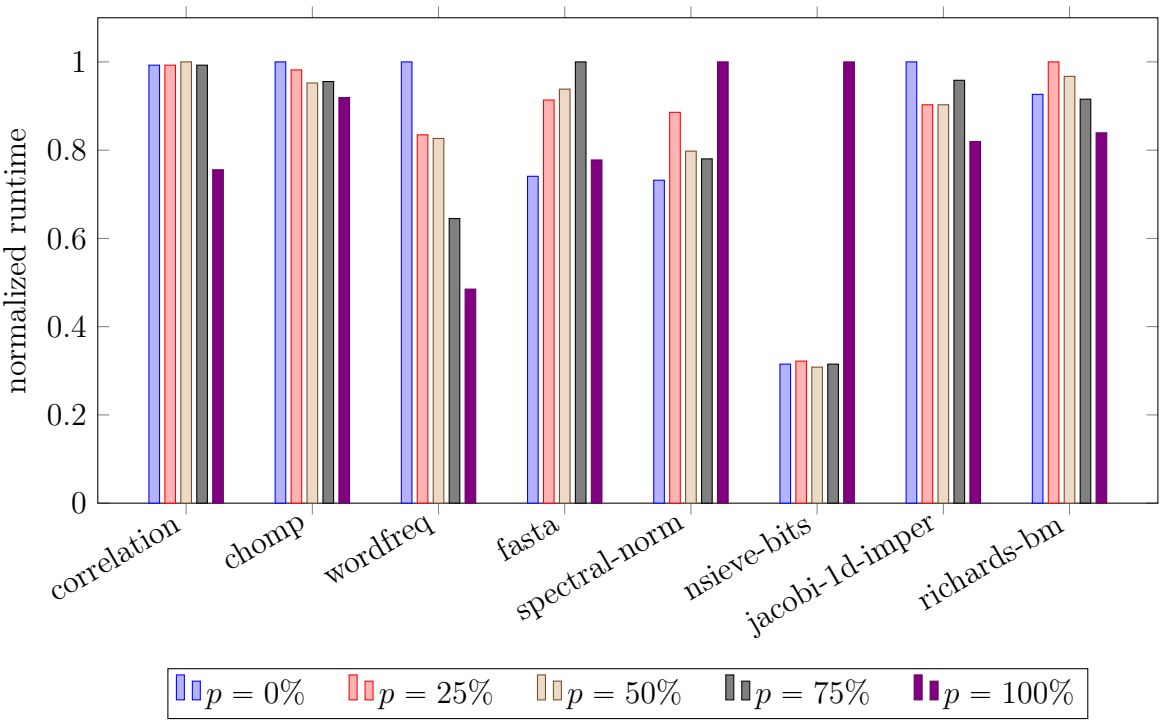


Figure 5.1: Runtimes of the SPRATTUS pass with the bilateral algorithm and a keep ratio of $p\%$. Runtimes are normalized to the maximal encountered runtime for each test.

the SPRATTUS pass to exceed the time limit. Further 34 benchmarks differed in their optimization time by a negligible amount for all of the configurations.

The optimization times of the remaining benchmarks closely resemble examples that are given in Figure 5.1:

(i) With 61 cases, the majority of the benchmarks show similar optimization times as `correlation`, `chomp` and `wordfreq`. This means $p = 100\%$ yields the shortest optimization time and the more elements are set to $\top$, the more time has to be spent in the optimization process.

(ii) Another six benchmarks provide similar results as `fasta`: The extremal values $p = 0\%$ and $p = 100\%$ yield the smallest optimization times whereas for values of p between those extremal points, the optimization time is increased.

(iii) The inverse case to (i) also occurs for six benchmarks: As depicted for the `spectral-norm` benchmark, preventing as many components as possible from being set to $\top$ unnecessarily extends the optimization time here.

(iv) A situation like for `nsieve-bits` is reached for four of the benchmarks: all $p \in \{0\%, 25\%, 50\%, 75\%\}$ perform with an almost identical optimization time whereas $p = 100\%$ dramatically increases the necessary amount of time.

(v) The remaining six benchmarks behave similar to the cases `jacobi-1d-imper` and `richards-benchmark`: Best performance is achieved with $p = 100\%$, the optimization requires most time for one of the non-extremal values $\{25\%, 50\%, 75\%\}$ for p.

Overall, this leads to the following conclusion: For the considered cases, $p = 100\%$, i.e. keeping as many components as possible, yields the best performance on average. Notice that this is the abstract consequence operation that resembles the one proposed in [Thakur, 2014] least.

5.1.2 Symbolic Abstraction Algorithms

For comparing the symbolic abstraction algorithms that are implemented in the SPRATTUS framework so far, one has to consider the simple unilateral algorithm $\widehat{\alpha}_{uni}$ as well as the incremental version $\widehat{\alpha}_{inc}$ of the unilateral algorithm that is optimized with respect to solver-specific reusage of computed information and the bilateral algorithm $\widehat{\alpha}_{bi}$.

In Figure 5.2, the respective results for the different symbolic abstraction algorithms for some representative test cases are depicted. Results for $\widehat{\alpha}_{bi}$ are gathered with $p = 100\%$, the configuration that yields the best average results on the considered benchmarks.

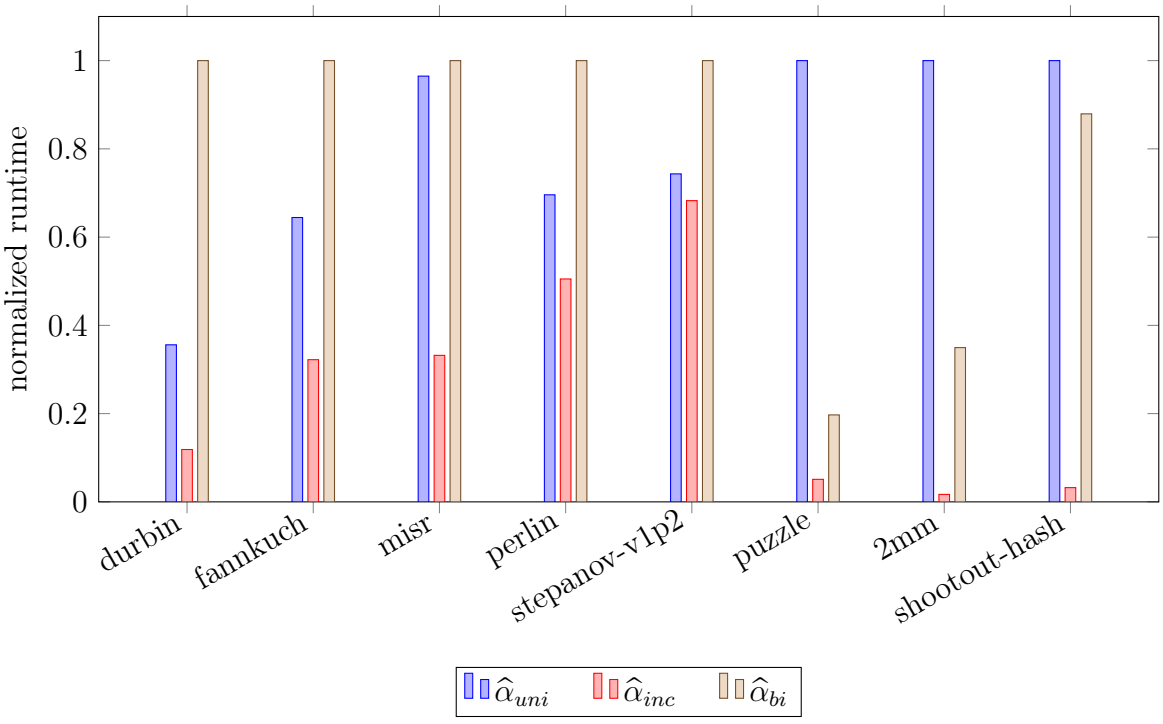


Figure 5.2: Runtimes of the SPRATTUS pass with the different symbolic abstraction algorithms. Runtimes are normalized to the maximal encountered runtime for each test.

For 25 of the 126 benchmarks, the optimization times are equal. For all benchmarks, for which the incremental unilateral algorithm did not require too much memory (that is all but six) it terminates fastest. Of these cases, on one hand, 75 are analyzed faster with the non-incremental unilateral algorithm than with the bilateral algorithm. Their achieved optimization times are similar to those of one of the first five entries in Figure 5.2. On the other hand, 20 further benchmarks are cases where the bilateral algorithm computes the correct result faster than the non-incremental unilateral algorithm as depicted in the remaining entries of Figure 5.2.

For the considered set of benchmarks, the incremental algorithm only crashed because of insufficient memory in cases in which both the unilateral and the bilateral algorithm would exceed the established time limit.

So, on this set of benchmarks and with these configurations, the incremental unilateral algorithm always performs significantly better than the other two algorithms. Of these, the non-incremental unilateral algorithm often outperforms the bilateral algorithm.

5.2 Fragment Decompositions

The specified fragment decomposition strategy can have an impact on both, the analysis results and the runtime of the analyzer. The decomposition strategies under evaluation are **Edges**, **Headers**, **Bodies** and **Backedges** as described in subsection 4.2.3. The analyses are performed with a product domain that contains the *Constant Propagation* domain and the *Restricted Equality* domain to obtain results for both transformations of the Sprattus pass.

Of the 126 benchmarks under test, 79 are analyzed without discovering an opportunity for optimization for any of the configurations. Moreover, the analysis of 10 further benchmarks is not successful due to the time limit. For the remaining benchmarks, the Sprattus pass is able to prove transformations applicable. Some characteristic examples for these cases are given in Table 5.1.

Of the remaining 37 benchmarks,

(i) for 18, the four configurations only compute two different results: a weaker result with fewer applicable transformations for **Edges** and for every other decomposition strategy one and the same stronger result. Examples for this are `exptree` and `ary3`.

(ii) For another nine benchmarks, the fragment decomposition had no influence on the number of transformations (cf. `dry`, `puzzle`).

(iii) Three benchmarks had strictly stronger results with the **Headers** strategy than with any other strategy.

Examples like `sphereflake`, `chomp` and `moments` trigger time-outs only for strategies with larger fragments.

In analysis time, the benchmarks follow patterns like they are visualized in Figure 5.3. Notable trends here are that an analyzer with the **Edges** strategy usually terminates

	Edges		Headers		Bodies		Backedges	
	Consts	Eqs	Consts	Eqs	Consts	Eqs	Consts	Eqs
BenchmarkGame/puzzle	1	4	1	4	1	4	1	4
McGill/exptree	0	8	0	9	0	9	0	9
McGill/misr	0	0	0	1	0	1	0	1
Misc-C++/sphereflake	0	1	0	1	—	—	0	1
Misc/whetstone	13	0	14	3	13	0	13	0
Shootout-C++/ary	1	4	1	5	1	5	1	5
Shootout-C++/moments	14	0	—	—	—	—	—	—
Shootout/ary3	22	2	22	3	22	3	22	3
Shootout/lists	1	3	1	3	14	3	1	3

Table 5.1: Examples for transformations performed for the supported fragment decompositions. The "Consts" columns contain the numbers of variable uses that have been replaced with constants or `undef` values by Constant Replacement. "Eqs" columns contain the numbers of variable uses that have been replaced with uses of other variables by Redundant Computation Elimination. Empty entries are due to time-outs.

faster than with other strategies (cf. `puzzle`, `ary`, `moments`). Furthermore, the analysis time for **Bodies** is often among the largest of the evaluated configurations (like in `sphereflake`, `whetstone` and `shootout-ary3`). However, there are examples which do not fulfill these criteria like `exptree` and `shootout-lists`.

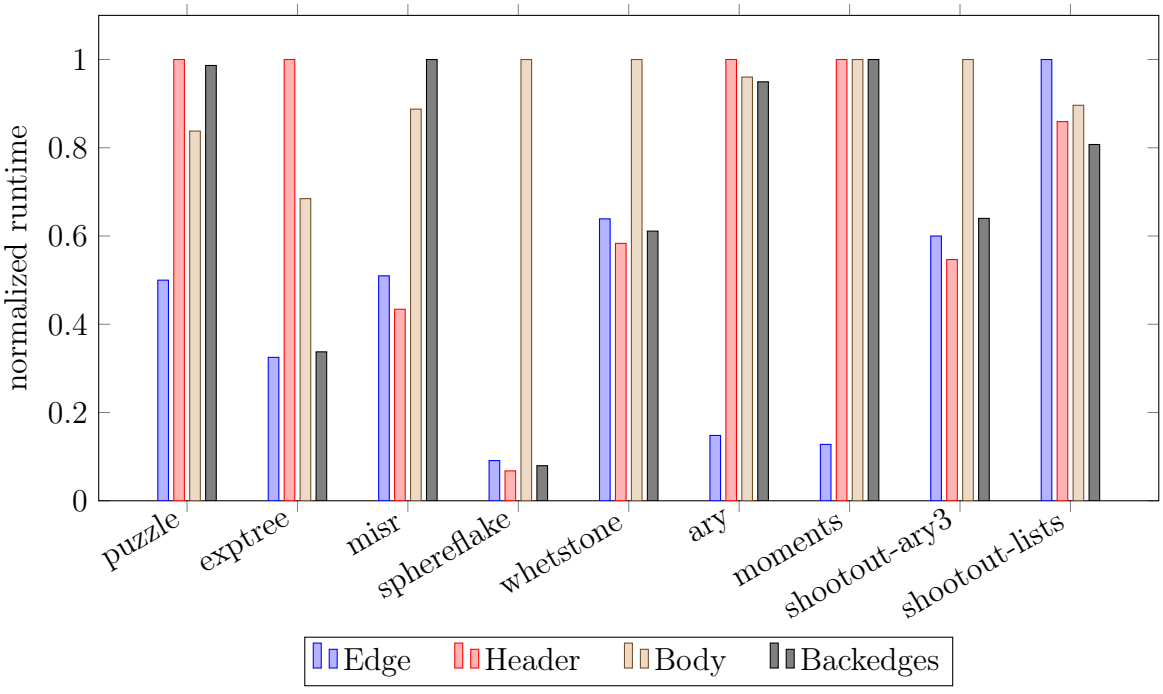


Figure 5.3: Runtimes of the SPRATTUS pass with different fragment decomposition strategies. They are normalized to the maximal encountered runtime for each test.

5.3 Abstract Domains

A key parameter in the configuration of SPRATTUS is the choice of the abstract domain (cf. section 4.4). In section 4.3, it has been shown that the number of issued calls to the SMT solver and hence the runtime of the analyzer depends quadratically on the height of the abstract domain in use. Therefore, differences in the height of the abstract domain can have potentially large impact on the runtime of the system.

SPRATTUS permits arbitrary combinations of abstract domains in reduced products, hence the space of evaluated configurations for this thesis has to be restricted to some sane domain combinations.

The configurations under evaluation are:

- *Constant Propagation* (C),
- *Constant Propagation* and *Equality to* `null` (C+EN),
- *Restricted Equality* (ER),
- *Constant Propagation* and *Restricted Equality* (C+ER), and
- *Constant Propagation*, *Equality to* `null` and *Restricted Equality* (C+EN+ER).

They are tested with the incremental unilateral algorithm.

Here, 81 benchmarks are not modified by the SPRATTUS pass. Table 5.2 gives examples from the remaining benchmarks.

Analysis results that often occur are:

(i) For the larger benchmarks, the analysis only terminates within the time limit for simple configurations such as C or C+EN (overall 11 cases, cf. `lpbench`, `oourafft`).

(ii) Ten of the 34 remaining benchmarks that are subject to actual transformations just have the same set of redundant computations for each configuration that contains the *Restricted Equality* domain (e.g. `fannkuch`).

(iii) Similarly, benchmarks like `whetstone` only provide optimization opportunities for the Constant Replacement transformation.

(iv) Moreover, 11 benchmarks can take advantage of the *Equality to* `null` domain in the product, like `nestedloop`, `queens` or `chomp`.

Considering analysis time, most benchmarks yield results that fulfill the following restrictions, as displayed for some examples in Figure 5.4:

(i) C+EN needs slightly more time for terminating than C.

(ii) C+ER or C+EN+ER are most time-consuming.

(iii) C+ER needs more time than C and than ER. However, the sum of the analysis times of both analyses in separate usually exceeds the time consumption of C+ER.

	C	C+EN	ER	C+ER		C+EN+ER	
	Consts	Consts	Eqs	Consts	Eqs	Consts	Eqs
BenchmarkGame/fannkuch	0	0	3	0	3	0	3
CoyoteBench/lpbench	29	—	—	—	—	—	—
Dhrystone/dry	7	7	1	7	1	7	1
McGill/chomp	0	3	37	0	38	3	38
McGill/queens	0	9	0	0	0	9	0
Misc/oourafft	75	75	—	—	—	—	—
Misc/whetstone	13	13	0	13	0	13	0
Shootout-C++/ary	1	1	4	1	4	1	4
Shootout-C++/lists	0	3	0	2	0	3	0
Shootout-C++/wordfreq	0	1	2	0	2	1	2
Shootout/ary3	22	23	2	22	2	23	2
Shootout/lists	1	1	0	1	3	1	3
Shootout/nestedloop	0	25	4	0	4	25	0

Table 5.2: Transformations performed for different abstract domains. The "Consts" columns contain the numbers of variable uses that have been replaced with constants or `undef` values by Constant Replacement. "Eqs" columns contain the numbers of variable uses that have been replaced with uses of other variables by Redundant Computation Elimination.

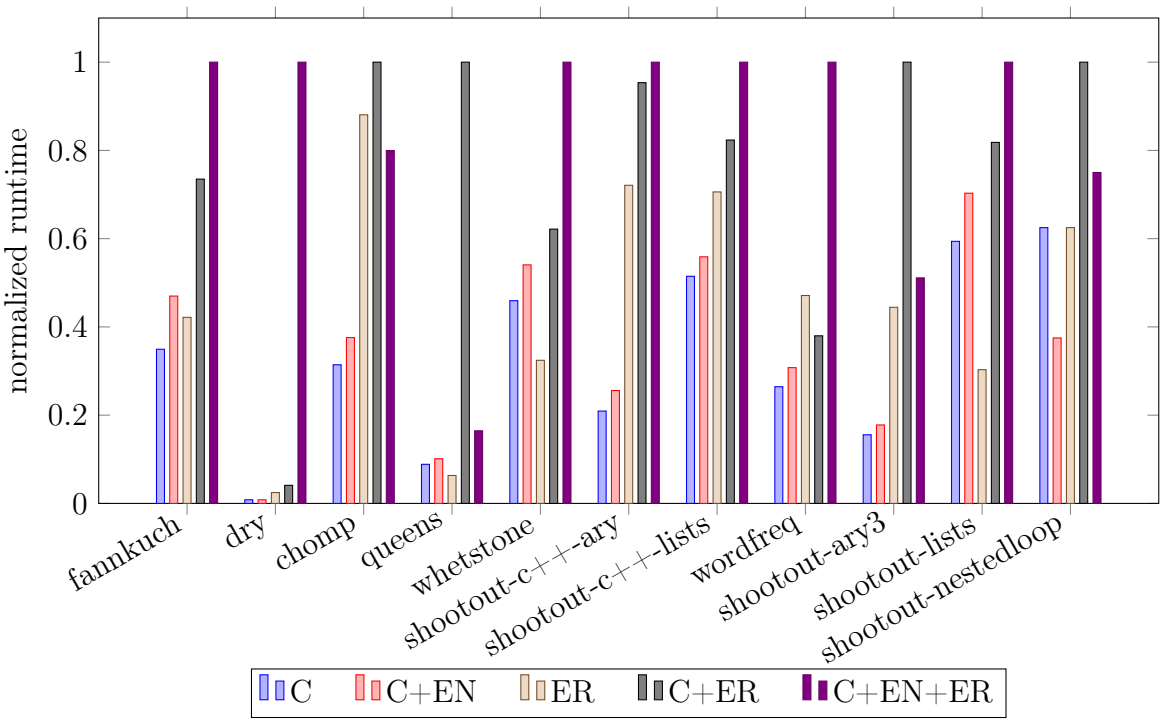


Figure 5.4: Runtimes of the SPRATTUS pass with different abstract domains. Runtimes are normalized to the maximal encountered runtime for each test.

5.4 Combining Compiler Optimizations

There exist publications on the topic of combined compiler optimizations which substantiate their work with examples for cases where the additional effort is profitable, but it is not investigated whether these cases are common in practical programming or they are just exceptional corner cases that seldom occur.

An example for such a publication is [Click and Cooper, 1995], where the small program that is referenced at different points of this thesis is used to motivate for the combination of program analyses and transformations. The above presented results give a measure of how often such cases occur in the considered benchmarks: Click and Cooper compare the performance of a constant propagation domain (comparable with the *Constant Propagation* configuration C) and a scalar congruence domain (which expresses facts that are similar to the information that the *Restricted Equality* domain ER provides) with the results from the reduced product thereof. Table 5.3 summarizes the evaluation results for the corresponding configurations of the SPRATTUS pass.

	C	ER	C+ER	
	Consts	Eqs	Consts	Eqs
McGill/chomp	0	37	0	38
Shootout-C++/lists	0	0	2	0
Shootout/lists	1	0	1	3

Table 5.3: Entries from Table 5.2 for which the reduced product of *Constant Propagation* and *Restricted Equality* is beneficial. The values that differ between separate analysis and the application of the reduced product are highlighted.

Of the 34 cases described in section 5.3 for which the analysis was able to prove transformations to be correct, only three could benefit from the reduced product of *Constant Propagation* domain and *Restricted Equality* domain. Moreover, the actual improvements in the number of performed transformations in these examples are very small.

Overall, for these benchmarks, the combined compiler optimization from [Click and Cooper, 1995] could not be proven generally useful. There exist only a few cases where the results differ from analyses in separate. However, for many of the benchmarks, the analysis time costs of computing results for both analyses separately exceed the time that is required for performing the corresponding analyses together. This way, combining analyses can be beneficial by reducing the overall runtime of the whole optimization process without improving precision significantly.

6 Related Work

Besides literature about the concept of symbolic abstraction, which is discussed in detail in section 2.3, related work can be classified into the use of decision procedures for satisfiability problems such as SMT in program analysis on one hand and work on the combination of program analyses, especially in the field of compiler optimization, on the other hand.

6.1 Satisfiability in Program Analysis

With increasing availability of computational power and efficient decision procedures for certain satisfiability problems, the use of automatic theorem provers for finding and proving properties of programs became more and more fashionable within the last decade.

Several examples for this trend are developed at Microsoft Research. These tools use the Z3 theorem prover to check or prove different aspects of program correctness. For example, the SLAM[1] project is used to check compliance with critical conditions of interfaces. Its results improve the performance of verifiers for hardware drivers.

Another example from Microsoft's SMT-based software engineering instruments is HAVOC[2], a tool for checking properties of potentially unsafe operations in the C programming language in the context of pointer manipulations, casts, and memory allocation including invariants of linked lists and arrays.

Notice that both of these tools are model checkers, i.e. in contrast to Sprattus, they are in general not able to handle arbitrary loops and derive loop invariants.

In principle, Sprattus also provides the tools for applications like the aforementioned Microsoft Research projects, i.e. for verifying arbitrary complex invariants of low-level programs. However, significant additional effort might be necessary to develop the necessary abstract domains.

In the LLVM project, there is also an application of SMT-based algorithms for software development: the KLEE[3] symbolic execution engine uses an SMT solver to automatically create high coverage test cases. KLEE generates traces of possible program runs of LLVM IR code encoding them in SMT formulas and identifies those traces that can lead to erroneous behavior. Some of the mechanics that are necessary for Sprattus have already been implemented similarly in KLEE, e.g. the SMT encoding of LLVM

[1] `http://research.microsoft.com/en-us/projects/slam/`
[2] `http://research.microsoft.com/en-us/projects/HAVOC/`
[3] `https://klee.github.io/`

IR to gather valid execution traces. The algorithm used for KLEE's test generation has also some aspects in common with the symbolic abstraction algorithms used in SPRATTUS. Still, the SPRATTUS framework provides a more general functionality for program analysis as it is not only limited to test generation. An application of SPRATTUS in the field of automatic test generation, in turn, would be conceivable as it could be used to generate preconditions that suffice to reach unintended program states.

The ALIVE project [Lopes et al., 2015] is an approach aiming at improving the correctness of compiler transformations in the LLVM framework. It provides a specification language for local rewrite rules, so called *peephole optimizations*. Besides simplifying the implementation of such rewrite rules, ALIVE also permits verifying the correctness of the specified transformations with the Z3 theorem prover.

For this purpose, just like SPRATTUS, ALIVE requires an SMT semantics for the supported LLVM instructions. However, ALIVE has strong restrictions with respect to the locality of the instructions in comparison to the formula generation of SPRATTUS. For example, ALIVE is neither able to handle loops nor to analyze connected code sequences over the boundaries of basic blocks as no branch instructions are supported.

6.2 Analysis Cooperation

Especially with a look at the compiler optimization setting, Click and Cooper describe the notion of a combined analysis that is stronger than the sum of its components in [Click and Cooper, 1995]. They describe the reduced product of the constant propagation analysis, dead code analysis and global value numbering with manually designed transformers. They motivate their work with an example program where this product is able to proof facts that none of the components could proof on its own. Yet, they do not investigate the relevance of such examples in practice.

A more generalized approach to the construction of reduced products for abstract domains in the restricted setting of logical programs is established in [Codish et al., 1995]. It requires an analysis designer to implement a `reduce` function that removes redundant information from the involved abstract states for every combination of domains that is created. They compare their generated domain products with existing hand-made combinations of the same domains on a standard set of benchmarks. Their generated combinations are always at least as good as the hand-made ones, for some domains the generated combinations perform even significantly better.

The ASTRÉE static analyzer [Cousot et al., 2005] is a commercial and industrially used program analysis tool that makes use of the combination of several abstract domains. It achieves this, as described in [Cousot et al., 2007], by hierarchically interacting domains. This hierarchy is constructed from simple abstract domains with a binary approximate reduced product operator. The values in the operand abstract domains of this product operator are sequentially evaluated in the analysis. Here, the operations in the secondly

evaluated abstract domain can benefit from the results computed in the firstly evaluated domain.

To allow domain cooperation, the abstract domain has to explicitly provide interaction mechanisms using a message system to refine the own information and to propagate gathered information to other domains.

The abstract transformer functions used in ASTRÉE are handcrafted and not necessarily optimal, however this enables ASTRÉE to provide its results with significantly smaller run time compared to SPRATTUS.

7 Conclusion

This thesis describes the realization of an application of the Sprattus framework in the field of compiler optimizations. Despite its shortcomings at competing in runtime with handcrafted implementations of common program transformations, the powerful interface and the flexibility in configuration presented Sprattus as a useful tool for rapid prototyping of new program analyses and transformations.

The set of program analyzer configurations that Sprattus supports has been extended by a bilateral symbolic abstraction algorithm as proposed in [Thakur, 2014]. The implementation was used to evaluate the performance of the bilateral algorithm in comparison to two versions of the unilateral algorithm. For the considered cases, one instance of the unilateral algorithm that made use of solver-internal features outperformed the bilateral algorithm.

Furthermore, new abstract domains have been introduced to the Sprattus framework. These include an implementation of the well-known intervals analysis and a domain that captures the truth value of an arbitrary unary or binary predicate over SSA-variables. For the latter, instantiations with applications in program optimization have been presented.

Moreover, the existing domains have been augmented with the functionality that is necessary for supporting the bilateral algorithm, especially the product abstract domain has been supplemented with a non-trivial abstract consequence operation that on average in the benchmarks performed better than the one described in [Thakur, 2014].

The developed compiler transformation pass was able to discover opportunities for program optimization that a default LLVM-based compiler could not perform. This gives an idea of potential uses of Sprattus-based compiler optimizations with specifically crafted domains for highly optimized release builds of larger software projects or for small embedded software.

However, no evidence for the profitability of combined compiler transformations as proposed in [Click and Cooper, 1995] could be found during the evaluation of the approach on a part of the LLVM benchmark suite.

The results of this thesis provide promising perspectives for the Sprattus framework. Although the approach entails drawbacks in the runtime behavior of the analysis, the thesis showed that it is applicable to real-world problems.

Besides the work on general purpose compiler transformations, many further applications of the framework are conceivable. This includes the implementation of highly

domain specific compiler optimizations as well as the verification of complex program properties. Current work on the project is concerned with memory safety and techniques to further improve the overall performance of Sprattus.

Bibliography

[Alpern et al., 1988] Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA. ACM.

[Barrett et al., 2009] Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2009). Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885.

[Biere et al., 2009] Biere, A., Heule, M., van Maaren, H., and Walsh, T. (2009). Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153.

[Buchwald, 2015] Buchwald, S. (2015). Optgen: A generator for local optimizations. In *Compiler Construction*, pages 171–189. Springer.

[Click and Cooper, 1995] Click, C. and Cooper, K. D. (1995). Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196.

[Codish et al., 1995] Codish, M., Mulkers, A., Bruynooghe, M., de la Banda, M. G., and Hermenegildo, M. (1995). Improving abstract interpretations by combining domains. *ACM Trans. Program. Lang. Syst.*, 17(1):28–44.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA. ACM.

[Cousot and Cousot, 1979] Cousot, P. and Cousot, R. (1979). Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA. ACM.

[Cousot et al., 2005] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2005). The astrée analyzer. In Sagiv, M., editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer Berlin Heidelberg.

[Cousot et al., 2007] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2007). Combination of abstractions in the astrée static analyzer. In Okada, M. and Satoh, I., editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer Berlin Heidelberg.

[Cytron et al., 1991] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

[de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In Ramakrishnan, C. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg.

[ISO 9899:2011, 2011] ISO 9899:2011 (2011). Programming languages - C. Standard, International Organization for Standardization, Geneva, CH.

[Kildall, 1973] Kildall, G. A. (1973). A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA. ACM.

[Kovásznai et al., 2012] Kovásznai, G., Fröhlich, A., and Biere, A. (2012). On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *SMT@ IJCAR*, pages 44–56.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California.

[Lopes et al., 2015] Lopes, N. P., Menendez, D., Nagarakatte, S., and Regehr, J. (2015). Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 22–32, New York, NY, USA. ACM.

[Nielson et al., 1999] Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of program analysis*. Springer.

[Reps et al., 2004] Reps, T., Sagiv, M., and Yorsh, G. (2004). Symbolic implementation of the best transformer. In Steffen, B. and Levi, G., editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 252–266. Springer Berlin Heidelberg.

[Seidl et al., 2012] Seidl, H., Wilhelm, R., and Hack, S. (2012). *Compiler Design: Analysis and Transformation*. Springer Science & Business Media.

[Thakur, 2014] Thakur, A. V. (2014). *Symbolic Abstraction: Algorithms and Applications*. dissertation, University of Wisconsin-Madison.

[Wegman and Zadeck, 1991] Wegman, M. N. and Zadeck, F. K. (1991). Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210.