

SMT-Based and Disjunctive Relational Abstract Domains for Static Analysis

by

Junjie Chen

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
May 2015

Professor Patrick Cousot

© Junjie Chen

All Rights Reserved, 2015

Dedication

This dissertation is dedicated to my parents and my wife.

Acknowledgements

First of all, I would like to thank deeply my advisor, Professor Patrick Cousot. He is the one who brought me into the fascinating field of Abstract Interpretation. It is a great fortune to have him to be my advisor. This thesis will not exist without his support and encouragement.

Second, I want to thank all of my committee members for their time and useful feedback: Patrick Cousot, Thomas Wies, Pietro Ferrara, Benjamin Goldberg and Kedar Namjoshi.

Personally, I want to thank Tim King for his patience in answering so many questions about the SMT solvers. I also want to thank Wei Wang for always willing to discuss with me and being a great friend.

I am indebted to my parents. As their son, I am so regretful that I cannot return them much.

Last but not least, I wish to thank my wife, Lei Xu. I cannot imagine even a day without her company.

This material is based upon work supported in part by the National Science Foundation under Grant No. 0926166.

Abstract

Abstract Interpretation is a theory of sound approximation of program semantics. In recent decades, it has been widely and successfully applied to the static analysis of computer programs. In this thesis, we will work on abstract domains, one of the key concepts in abstract interpretation, which aim at automatically collecting information about the set of all possible values of the program variables. We will focus, in particular, on two aspects: the combination with theorem provers and the refinement of existing abstract domains.

Satisfiability modulo theories (SMT) solvers are popular theorem provers, which proved to be very powerful tools for checking the satisfiability of first-order logical formulas with respect to some background theories. In the first part of this thesis, we introduce two abstract domains whose elements are logical formulas involving finite conjunctions of affine equalities and finite conjunctions of linear inequalities. These two abstract domains rely on SMT solvers for the computation of transformations and other logical operations.

In the second part of this thesis, we present an abstract domain functor whose elements are binary decision trees. It is parameterized by decision nodes which are a set of boolean tests appearing in the programs and by a numerical or symbolic abstract domain whose elements are the leaves. This new binary decision tree abstract domain functor provides a flexible way of adjusting the cost/precision ratio in path-dependent static analysis.

Contents

Dedication	iii
Acknowledgements	iv
Abstract	v
List of Tables	xi
Introduction	1
I Background	4
1 Abstract Interpretation	5
1.1 Elements of Abstract Interpretation	5
1.1.1 Notations	5
1.1.2 Abstractions and Concretizations	7
1.1.3 Abstract and Concrete Transformations	9
1.1.4 Fixpoint Computation	11
1.1.5 Reduced Product	13
1.1.6 Abstract Domains	15
1.2 A Simple Programming Language	16

1.2.1	Abstract Syntax	16
1.2.2	States	18
1.2.3	Traces	18
1.2.4	Trace Semantics of a Simple Programming Language	19
1.2.5	Action Path Semantics: An Example of Abstract Interpretation	23
1.3	Numerical Static Analysis	28
1.3.1	Discovering properties of numerical variables	29
1.3.2	Intervals	30
1.3.3	Polyhedra	33
2	Satisfiability Modulo Theories	35
2.1	First-Order Logic	35
2.1.1	Syntax	36
2.1.2	Interpretations	37
2.1.3	Theories and Models	38
2.2	The SMT Problem	39
2.2.1	Satisfiability and Validity	39
2.2.2	Theories of interest	41
2.2.3	Abstract DPLL(\mathcal{T})	43
2.2.4	Combining Theories	45
2.3	SMT-LIB	47
2.3.1	Theories	47
2.3.2	Logics	49
2.3.3	SMT2 Commands	51

II	SMT-based Abstract Domains	53
3	Affine Equalities	54
3.1	Normal Form of Affine Equalities	54
3.1.1	Generation from Points P	55
3.1.2	Normalization	59
3.2	SMT-Based Affine Equality Abstract Domain	61
3.2.1	Representation	61
3.2.2	Binary Operations	62
3.2.3	Transfer Functions	64
3.2.4	Example	66
3.2.5	Implementation	67
4	Linear Inequalities	70
4.1	Representation	70
4.1.1	Simplification	71
4.2	Binary Operations	72
4.2.1	Inclusion Testing	72
4.2.2	Equality Testing	73
4.2.3	Meet	74
4.2.4	Join	74
4.3	Transfer Functions	77
4.3.1	Assignment	78
4.3.2	Test	80
4.4	Widening	81
4.5	Example	82
4.6	Implementation	83

4.7	Related Work	85
III	Binary Tree-based Abstract Domains	87
5	Branch Condition Path Abstraction	88
5.1	Branch Condition Graph	88
5.2	Branch Condition Path Abstraction	90
5.2.1	Condition Path Abstraction	91
5.2.2	Loop Condition Elimination	92
5.2.3	Duplication Elimination	94
5.2.4	Branch Condition Path Abstraction	95
6	Binary Decision Tree Abstract Domain Functor	97
6.1	Introduction	97
6.2	Binary Operations	102
6.2.1	Inclusion and Equality	102
6.2.2	Meet and Join	103
6.3	Transfer Functions	105
6.3.1	Loop Test Transfer Function	105
6.3.2	Branch Test Transfer Function	105
6.3.3	Assignment Transfer Function	107
6.4	Extrapolation Operators	108
6.4.1	Widening	109
6.4.2	Narrowing	109
6.5	Other Operators	111
6.6	Example	112
6.7	Related Work	113

Conclusion	114
A Proof of Theorem 1.2.3	116
Bibliography	126

List of Tables

3.1	Comparing SMT-based affine equality abstract domain and Equality abstract domain in Apron abstract domain library	68
4.1	Comparing SMT-based linear inequality abstract domain and Polka abstract domain in Apron abstract domain library	84
4.2	Comparing the join operator in SMT-based linear inequality abstract domain and Polka abstract domain in Apron abstract domain library	85

Introduction

As computer systems and software grow more and more complex, how to ensure their correctness is becoming a very important problem and considered a great challenge. In recent years, more and more time and effort have been applied to find and eliminate bugs, even much more than the time and effort spent on writing programs. Moreover, as we rely more and more on software, the consequences of a bug are more and more dramatic, causing great financial and even human losses. In 1992, the cumulated imprecision errors in a Patriot missile defense caused a missile hit wrong target which resulting in 28 people being killed [Ske92]. Another extreme example is the failure of the Ariane 5 launcher in 1996 is caused by the overflow bug [ea96].

In real life, testing is a massively used technique to ensure the correctness of programs. Unfortunately, it is also very unreliable. This is because it only considers a very small sample of program behaviors which will easily leave bugs. Moreover, as software complexity grows exponentially with time, testing does not seem to catch up with it giving worse and worse results while becomes more and more costly. Formal methods, on the other hand, provide mathematical techniques to cover all program behaviors and use symbolic representations to achieve efficiency.

Static Analysis is the analysis of computer programs that is able to discover properties of the analyzed program without actually executing it. It should preferably always terminate in a predictable period. By Rice's theorem [Ric53], any property on the outcome

of a program that is not always true for all programs or false for all programs - which includes every single interesting property - is undecidable. Hence, it is not possible to write a program able to represent and to compute all possible executions of any program in all its possible execution environments which means a perfect static analyzer does not exist. Thus every static analyzer should make approximations, one way or another.

Abstract Interpretation [CC92b, CC77, CC79b] is a general theory of the sound approximations of program semantics. The program semantics refers to the mathematical meaning of computer programs. Here, sound approximation means that it guarantees the static analysis is correct and exhaustive. A false negative will never be yielded, but by undecidability false alarms (or false positive) may be produced. An abstract domain, a key concept in abstract interpretation, is a class of properties together with a set of operators to manipulate them, aiming at collecting information about the set of all possible values of the program variables. Unlike other formal methods, once an abstract domain is designed, the static analysis based on it is fully automatic and scalable. Moreover, abstract interpretation provides a unifying framework for expressing in different level of abstractions of several different seemingly unrelated program semantics, which are widely used in different formal methods including proof methods, model checking, type checking, type inference, and semantic-based static analysis.

In many different formal methods, the program behaviors are often encoded in formulas over a set of fixed first-order logic theories. Satisfiability Modulo Theory (SMT) solvers [BSST09] are often used to decide whether these first-order formulas are satisfiable or not within this set of fixed first-order logic theories. In this thesis, we are going to introduce logical abstract domains where program properties are represented by first-order logical formulas over first-order theories and rely on SMT solvers as back-end technology for the computation of transformations and other logical operations.

Commonly, the abstract domains use convex sets which are conjunctions of linear

constraints to represent program properties. Normally, the convexity of these abstract domains makes the analysis scalable (with exceptions, e.g., Polyhedra abstract domain has exponential complexity). On the other hand, the absence of disjunctions may cause rough approximations and produce much less precise results, gradually leading to false alarms or even worse to the complete failure to prove the desired program property. In this thesis, we will also introduce a refined binary decision tree abstract domain functor by adding controllable disjunctions to many already existing abstract domains to provide a flexible way of adjusting the cost/precision ratio in path-dependent static analysis.

This thesis is organized as follows. Chapter 1 recalls some key features and concepts of the abstract interpretation framework. A simple imperative programming language with its syntax and formal semantics has also been presented. Chapter 2 gives a brief introduction of satisfiability modulo theories, including the notion of first-order logic, $DPLL(\mathcal{T})$, Nelson-Oppen method and SMT-LIB. The first two chapters will serve throughout the following chapters. Chapter 3 introduces a logical abstract domain whose elements are logical formulas involving finite conjunctions of affine equalities. SMT solvers are used for the computation of transformations and other logical operations in the domain. Chapter 4 introduces another logical abstract domain which is able to represent and manipulate invariants by finite conjunctions of linear inequalities. SMT solvers are also used for the computation of transformations and other logical operations in the domain. Chapter 5 introduces the branch condition graph and defines the branch condition path abstraction. This chapter will also serve throughout the next chapter. Chapter 6 introduces an abstract domain functor whose elements are binary decision trees. It is parameterized by decision nodes which are a set of boolean tests appearing in the programs and by a numerical or symbolic abstract domain whose elements are the leaves.

Part I

Background

Chapter 1

Abstract Interpretation

Abstract Interpretation [CC92b] is a general theory of sound abstraction and approximation of program semantics. In this chapter, we first recall some of its key features and concepts. We then present a simple imperative programming language with its syntax and formal semantics in the abstract interpretation framework. We also discuss numerical program analysis using abstract interpretation framework and present two classical numerical abstract domains. The material in this chapter will be referenced later in this thesis.

1.1 Elements of Abstract Interpretation

We recall some core definitions and features of Abstraction Interpretation here. We refer the interested reader to [CC92b] for more details.

1.1.1 Notations

Partial orders. A *partial order* \sqsubseteq is a binary relation that is reflexive, transitive, and anti-symmetric. A non-empty set \mathbb{D} equipped with a partial order \sqsubseteq is called *partially*

ordered set (or *poset*). A reflexive, transitive but not anti-symmetric binary relation is called *preorder*. A non-empty set \mathbb{D} equipped with a preorder can be easily turned into a poset by identifying elements in \mathbb{D} such that both $a \sqsubseteq b$ and $b \sqsubseteq a$. An *infimum* \perp is the least element in poset and a *supremum* \top is the largest element in a poset. Given a pair of elements $a, b \in \mathbb{D}$, the *least upper bound* (also called *join*) is denoted $a \sqcup b$, and the *greatest lower bound* (also called *meet*) is denoted $a \sqcap b$. Note that, the infimum and supremum, the least upper bound and greatest lower bound do not always exist. But if any of them exists, it's unique (by anti-symmetric of \sqsubseteq). A *complete partial order* (or *cpo*) is a poset $(\mathbb{D}, \sqsubseteq)$ such that any increasing chain C of elements of \mathbb{D} has a least upper bound $\sqcup C$ in \mathbb{D} (we denote the join and meet of $D \subseteq \mathbb{D}$ respectively as $\sqcup D$ and $\sqcap D$). Note that an infimum always exists in cpo due to $\perp = \sqcup \emptyset$. A *lattice* is a poset $(\mathbb{D}, \sqsubseteq)$ with an infimum \perp and a supremum \top in \mathbb{D} , and least upper bound \sqcup and greatest lower bound \sqcap for every pair of elements in \mathbb{D} . A *complete lattice* is a lattice such that any subset $D \subseteq \mathbb{D}$ has a least upper bound $\sqcup D$. Note that, a complete lattice is always a cpo.

Fixpoints. The set of *fixpoints* of a function $f \in \mathbb{D} \rightarrow \mathbb{D}$ is $\text{fp}(f) \triangleq \{x \in \mathbb{D} \mid f(x) = x\}$. We let $\text{fp}_a(f)$ be the set of fixpoints of f greater than or equal to a . The set of *post-fixpoints* of a function $f \in \mathbb{D} \rightarrow \mathbb{D}$ on a poset $(\mathbb{D}, \sqsubseteq)$ is $\text{postfp}(f) \triangleq \{x \in \mathbb{D} \mid f(x) \sqsubseteq x\}$ with $\text{postfp}_a(f) \triangleq \{x \in \mathbb{D} \mid a \sqsubseteq x \wedge f(x) \sqsubseteq x\}$. The set of *pre-fixpoints* of a function $f \in \mathbb{D} \rightarrow \mathbb{D}$ on a poset $(\mathbb{D}, \sqsubseteq)$ is $\text{prefp}(f) \triangleq \{x \in \mathbb{D} \mid x \sqsubseteq f(x)\}$ with $\text{prefp}_a(f) \triangleq \{x \in \mathbb{D} \mid x \sqsubseteq a \wedge x \sqsubseteq f(x)\}$. We write $\text{lfp}(f)$ for the least fixpoint of the function f , and $\text{gfp}(f)$ for the greatest fixpoint, if such fixpoints exist. Moreover, $\text{lfp}_a(f)$ is the least fixpoint of f greater than or equal to a , while $\text{gfp}_a(f)$ is the greatest fixpoint of f smaller than or equal to a . We then recall two fundamental theorems about fixpoints of functions in ordered structures:

Theorem 1.1.1 (Tarski's fixpoints [Tar55]). *The set of fixpoints of a monotonic function*

$f \in \mathbb{D} \xrightarrow{\sqsubseteq} \mathbb{D}$ on a complete lattice is a complete lattice. Moreover,

$$\begin{aligned} \text{lfp}(f) &= \sqcap \text{postfp}(f) = \sqcap \{x \in \mathbb{D} \mid f(x) \sqsubseteq x\} \\ \text{gfp}(f) &= \sqcup \text{prefp}(f) = \sqcup \{x \in \mathbb{D} \mid x \sqsubseteq f(x)\} \end{aligned}$$

□

Theorem 1.1.2 (Kleene’s fixpoints [Kle52]). *Let a continuous function $f \in \mathbb{D} \xrightarrow{\sqsubseteq} \mathbb{D}$ on a cpo $(\mathbb{D}, \sqsubseteq, \sqcup)$ and $a \in \mathbb{D}$ be a pre-fixpoint for f , then $\text{lfp}_a(f) = \sqcup \{f^i(a) \mid i \in \mathbb{N}\}$. Dually, let $a \in \mathbb{D}$ be a post-fixpoint for f , then $\text{gfp}_a(f) = \sqcap \{f^i(a) \mid i \in \mathbb{N}\}$.* □

Tarski’s fixpoints theorem states that any monotonic function in a complete lattice has least and greatest fixpoints. Kleene’s fixpoints theorem states that those fixpoints are computable iteratively. More details can be found in [CC79a].

1.1.2 Abstractions and Concretizations

Let $(\mathbb{D}^b, \sqsubseteq^b)$ and $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$ be two posets used as semantic domains which is a set of elements carrying the information about programs. $d \sqsubseteq d'$ in a semantic domain means that d carries less information than d' . A monotonic function $\gamma \in \mathbb{D}^\sharp \rightarrow \mathbb{D}^b$ is called *concretization function* when each element $d^\sharp \in \mathbb{D}^\sharp$ represents some information $\gamma(d^\sharp) \in \mathbb{D}^b$. We call $(\mathbb{D}^b, \sqsubseteq^b)$ the *concrete domain* while $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$ the *abstract domain*. We say that $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$ is an abstraction of $(\mathbb{D}^b, \sqsubseteq^b)$. A monotonic function $\alpha \in \mathbb{D}^b \rightarrow \mathbb{D}^\sharp$ can be defined, which is called *abstraction function*, to specify which abstract element $\alpha(d^b) \in \mathbb{D}^\sharp$ can be safely used to represent a concrete element $d^b \in \mathbb{D}^b$.

In [CC77], Cousot and Cousot introduced *Galois connection* between $(\mathbb{D}^b, \sqsubseteq^b)$ and $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$ by the function pair (α, γ) such that:

Definition 1.1.1 (Galois connection).

$$\forall x^b \in \mathbb{D}^b, x^\# \in \mathbb{D}^\#, \alpha(x^b) \sqsubseteq^\# x^\# \iff x^b \sqsubseteq^b \gamma(x^\#)$$

□

This is often pictured as follows:

$$(\mathbb{D}^b, \sqsubseteq^b) \xrightleftharpoons[\alpha]{\gamma} (\mathbb{D}^\#, \sqsubseteq^\#)$$

As a consequence, we have:

$$\begin{aligned} \forall x^\# \in \mathbb{D}^\# : (\alpha \circ \gamma)(x^\#) \sqsubseteq^\# x^\# \\ \forall x^b \in \mathbb{D}^b : x^b \sqsubseteq^b (\gamma \circ \alpha)(x^b) \end{aligned}$$

Moreover, $\alpha(x^b)$ will be the best, which is the most precise, abstraction of x^b in $\mathbb{D}^\#$. When γ is injective, or equivalently, α is surjective, we have $\forall x^\# \in \mathbb{D}^\# : (\alpha \circ \gamma)(x^\#) = x^\#$, then the pair (α, γ) is called *Galois insertion*.

When $(\mathbb{D}^\#, \sqsubseteq^\#)$ has $\sqcap^\#$, it forms a so-called *Moore family* [CC79b] and the abstraction function α can be defined by the concretization function γ as $\alpha(x^b) \triangleq \sqcap^\#\{x^\# \mid x^b \sqsubseteq^b \gamma(x^\#)\}$. Dually, when $(\mathbb{D}^b, \sqsubseteq^b)$ has \sqcup^b , then the concretization function can be defined by the abstraction function α as $\gamma(x^\#) \triangleq \sqcup^b\{x^b \mid \alpha(x^b) \sqsubseteq^\# x^\#\}$. Note that, Moore family and Galois connection are equivalent.

In practice, a Galois connection does not always exist between the concrete and the abstract domains. In some cases no α function exists. In those circumstances, Cousot and Cousot explained in [CC92b] how to relax the Galois connection framework in order to work only with a concretization function γ , or dually, only with an abstraction function α . However, concretization-based abstract interpretation is much more used in practice.

1.1.3 Abstract and Concrete Transformations

Let $f^b \in \mathbb{D}^b \rightarrow \mathbb{D}^b$ be a concrete transformer on the concrete domain $(\mathbb{D}^b, \sqsubseteq^b)$, a transformer $f^\# \in \mathbb{D}^\# \rightarrow \mathbb{D}^\#$ on the abstract domain $(\mathbb{D}^\#, \sqsubseteq^\#)$ is called *sound abstract transformer* for f^b if and only if $\forall x^\# \in \mathbb{D}^\# : (f^b \circ \gamma)(x^\#) \sqsubseteq^b (\gamma \circ f^\#)(x^\#)$, or dually, $\forall x^b \in \mathbb{D}^b : (\alpha \circ f^b)(x^b) \sqsubseteq^\# (f^\# \circ \alpha)(x^b)$. It states that computing in the abstract always yields less or as much information as in the concrete.

When both α and γ exist, the *best abstract transformer* $f^\#$ of f^b can be defined as $f^\# \triangleq \alpha \circ f^b \circ \gamma$. Ideally, we would have $f^b \circ \gamma = \gamma \circ f^\#$. This means that the abstract computation does not lose any information with respect to the concrete one. We call $f^\#$ the *exact abstract transformer* of f^b . Unfortunately, exact abstract transformers seldom exist. However, when a Galois insertion exists between the concrete domain and the abstract domain, the exact abstract transformer do exist and it is also the best abstract transformer.

Note that all these definitions and properties apply to n-ary operators as well.

Let $f_1^b, f_2^b \in \mathbb{D}^b \rightarrow \mathbb{D}^b$ be two monotonic concrete transformers on the concrete domain $(\mathbb{D}^b, \sqsubseteq^b)$ and $f_1^\#, f_2^\# \in \mathbb{D}^\# \rightarrow \mathbb{D}^\#$ be two monotonic abstract transformers on the abstract domain $(\mathbb{D}^\#, \sqsubseteq^\#)$. $f_1^\#$ is the sound abstract transformer of f_1^b while $f_2^\#$ is the sound abstract transformer of f_2^b . Then the composition of $f_1^\# \circ f_2^\#$ is still a sound abstract transformer of the composition of $f_1^b \circ f_2^b$. Moreover, the composition of exact abstract transformers is still an exact abstract transformer, but the composition of best abstract transformers is not necessarily a best abstract transformer. Hence, the result of an analysis by combining a set of abstract transformers, even each abstract transformer is the best abstract transformer, could still be very poor (imprecise).

In abstract interpretation, a core principle is that all kinds of semantics are expressed as fixpoints of monotonic functions (or transformers) in posets. Given a semantics expressed

as $\text{lfp}_{a^b}(f^b)$ in concrete domain $(\mathbb{D}^b, \sqsubseteq^b)$, what we need is to abstract it as $\text{lfp}_{a^\sharp}(f^\sharp)$ in abstract domain $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$.

Let us first consider the case of $(\mathbb{D}^b, \sqsubseteq^b)$ and $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$ are linked by a Galois connection by the function pair (α, γ) . In [Cou02], Cousot presented the following two fixpoint transfer theorems:

Theorem 1.1.3 (Tarski's fixpoint transfer). *Let $(\mathbb{D}^b, \sqsubseteq^b)$ and $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$ are both complete lattice and $a \in \mathbb{D}^\sharp$, $f^b \in \mathbb{D}^b \rightarrow \mathbb{D}^b$ and $f^\sharp \in \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$ are two monotonic functions, then we have*

$$\text{lfp}_{\gamma(a)}(f^b) \sqsubseteq^b \gamma(\text{lfp}_a(f^\sharp)).$$

□

Theorem 1.1.4 (Kleene's fixpoint transfer). *Let $(\mathbb{D}^b, \sqsubseteq^b)$ and $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$ are both cpos and $a \in \mathbb{D}^\sharp$, $f^b \in \mathbb{D}^b \rightarrow \mathbb{D}^b$ and $f^\sharp \in \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$ are two monotonic functions, If γ is continuous, then*

$$\text{lfp}_{\gamma(a)}(f^b) \sqsubseteq^b \gamma(\text{lfp}_a(f^\sharp)).$$

Moreover, if f^\sharp is the exact abstract transformer, then

$$\text{lfp}_{\gamma(a)}(f^b) = \gamma(\text{lfp}_a(f^\sharp)).$$

□

More commonly, we may have $f^\sharp \neq \alpha \circ f^b \circ \gamma$ as the latter is too difficult to compute or does not exist at all (such as that there is no abstraction function α); or even worse where f^\sharp does not admit a least fixpoint (or any fixpoint at all). In all those cases, where no optimal fixpoint abstraction can be defined or computed, we abstract a concrete least fixpoint to an abstract post-fixpoint instead of a least fixpoint.

1.1.4 Fixpoint Computation

Assume we have designed an abstract domain where all elements are computer representable and all the abstract transformers we need are computable. We then need to consider how to efficiently compute fixpoints in the abstract domain.

When the elements in the abstract domain satisfy the *ascending chain condition* (i.e., no infinite strictly increasing chain), then Kleene's theorem (Theorem 1.1.2) gives a constructive way to compute the least fixpoint in the abstract domain:

Theorem 1.1.5. *Let $(\mathbb{D}^\#, \sqsubseteq^\#, \sqcup^\#)$ be a cpo and $f^\# \in \mathbb{D}^\# \rightarrow \mathbb{D}^\#$ is monotonic, $x^\#$ is a pre-fixpoint for $f^\#$, if $\mathbb{D}^\#$ has no infinite strictly increasing chain, then the Kleene iterations $x_{i+1}^\# \triangleq f^\#(x_i^\#)$ will converge in finitely many steps towards $\text{lfp}_{x^\#}(f^\#)$. \square*

One of the simplest case of theorem 1.1.5 is the *sign domain* (see [CC76]) which is a finite domain. A more complex case is the *constant domain* (see [Kil73]) which is infinite but satisfies the ascending chain condition.

For those domains that are containing infinite strictly increasing chain, normally theorem 1.1.5 can not guarantee the termination of iterations except some rare cases. In [SW04] and [RCK04], the authors introduce non-standard iteration schemes as well as sufficient conditions on the abstract functions for the iterates to converge in finite time within, respectively, the interval domain and the domain of varieties.

In [CC76], Cousot and Cousot introduced a more general way of dealing with domains with infinite strictly increasing chains, which is called *widening*.

Definition 1.1.2 (Widening). A widening $\nabla^\# \in (\mathbb{D}^\# \times \mathbb{D}^\#) \rightarrow \mathbb{D}^\#$ is satisfying:

- $\forall x^\#, y^\# \in \mathbb{D}^\# : x^\# \sqsubseteq^\# x^\# \nabla^\# y^\# \wedge y^\# \sqsubseteq^\# x^\# \nabla^\# y^\#$;
- for any sequence $(x_i^\#, i \in \mathbb{N})$, the sequence defined as $y_0^\# \triangleq x_0^\#$ and $y_{i+1}^\# \triangleq x_i^\# \nabla^\# y_i^\#$ converges in finite time.

□

The following theorem shows that we can compute the Kleene's fixpoint with finite iterations:

Theorem 1.1.6 ([CC92c]). *Let f^\sharp be a sound abstraction of f^b , $\gamma(x^\sharp)$ is a pre-fixpoint for f^b , then the sequence defined as $y_0^\sharp \triangleq x^\sharp$ and $y_{i+1}^\sharp \triangleq y_i^\sharp \nabla^\sharp f^\sharp(y_i^\sharp)$ reaches a stable iterate y_n^\sharp such that $f^\sharp(y_n^\sharp) \sqsubseteq^\sharp y_n^\sharp$ in finite time. Moreover, $\text{lfp}_{\gamma(x^\sharp)}(f^b) \sqsubseteq^b \gamma(y_n^\sharp)$.* □

Note that the output of the iterations with widening is generally not a monotonic function of x^\sharp . Thus, it does not make any guarantee on the precision of the computed approximation, but only ensures soundness and termination.

Sometimes, the result of widening $y_n^\sharp = y_n^\sharp \nabla^\sharp f^\sharp(y_n^\sharp)$ is a strict post-fixpoint of f^\sharp : $f^\sharp(y_n^\sharp) \sqsubset^\sharp y_n^\sharp$. Thus, the approximation y_n^\sharp can be refined by applying f^\sharp some more times without widening: $z_0^\sharp \triangleq y_n^\sharp, z_{i+1}^\sharp \triangleq f^\sharp(z_i^\sharp)$. Unfortunately, there is no guarantee that this process will terminate due to \mathbb{D}^\sharp may have infinite strictly decreasing chains. In [CC76], Cousot and Cousot also introduced a *narrowing* operator Δ to enforce the termination of such refinement.

Definition 1.1.3 (Narrowing). A narrowing $\Delta^\sharp \in (\mathbb{D}^\sharp \times \mathbb{D}^\sharp) \rightarrow \mathbb{D}^\sharp$ is satisfying:

- $\forall x^\sharp, y^\sharp \in \mathbb{D}^\sharp : x^\sharp \sqcap^\sharp y^\sharp \sqsubseteq^\sharp x^\sharp \Delta^\sharp y^\sharp \sqsubseteq^\sharp x^\sharp$;
- for any sequence $(x_i^\sharp, i \in \mathbb{N})$, the sequence defined as $y_0^\sharp \triangleq x_0^\sharp$ and $y_{i+1}^\sharp \triangleq y_i^\sharp \Delta^\sharp x_{i+1}^\sharp$ converges in finite time.

□

The following theorem shows that we can refine an approximate fixpoint using decreasing iterations with narrowing in finite time:

Theorem 1.1.7 ([CC92c]). *Let f^\sharp be a sound abstraction of f^b , if $\text{lfp}_{\gamma(x^\sharp)}(f^b) \sqsubseteq^b \gamma(y^\sharp)$, then the sequence defined as $z_0^\sharp \triangleq y^\sharp$ and $z_{i+1}^\sharp \triangleq z_i^\sharp \Delta^\sharp f^\sharp(z_i^\sharp)$ reach a stable iterate z_n^\sharp such that $z_{n+1}^\sharp = z_n^\sharp$ in finite time. Moreover, we have $\text{lfp}_{\gamma(x^\sharp)}(f^b) \sqsubseteq^b \gamma(z_n^\sharp) \sqsubseteq^b \gamma(y^\sharp)$. \square*

In fact, widening and narrowing are very general methods. They can also be useful in domains satisfying ascending chain condition or descending chain condition (i.e., no infinite strictly decreasing chain). In case when efficiency is much more important than precision, we can design such domains by computing a fixpoint abstraction with much fewer steps than the classical Kleene's iterations (theorem 1.1.5). As proved by Cousot and Cousot in [CC92c], more precision can always be obtained by using widening and narrowing on an abstract domain with infinite chains than by further abstracting this domain into a domain satisfying the ascending chain condition.

Recently, a new method called *policy iteration* (see [CGG⁺05], [GGTZ07] and [GS07]), which comes from Game Theory, has been studied in fixpoint computation. It divides the fixpoint computation into several simple steps that can be easily and efficiently calculated. Policy iteration can guarantee the termination of computation. Compare to widening / narrowing, policy iteration sometimes is more efficient and precise. Unfortunately, it cannot be applied to those popular relational abstract domains, such as Polyhedral, which limit its usage in practice.

1.1.5 Reduced Product

One of the most convenient ways to obtain more precision in the analysis by abstract interpretation is to combine several already existing domains into a new, more powerful one. Given a concrete domain $(\mathbb{D}^b, \sqsubseteq^b)$ and two abstract domains $(\mathbb{D}_1^\sharp, \sqsubseteq_1^\sharp)$ and $(\mathbb{D}_2^\sharp, \sqsubseteq_2^\sharp)$ with their concretizations $\gamma_1 : \mathbb{D}_1^\sharp \rightarrow \mathbb{D}^b$ and $\gamma_2 : \mathbb{D}_2^\sharp \rightarrow \mathbb{D}^b$, we can define the *product domain*:

$$\mathbb{D}^\sharp \triangleq \mathbb{D}_1^\sharp \times \mathbb{D}_2^\sharp$$

with ordering:

$$(x_1^\sharp, x_2^\sharp) \sqsubseteq^\sharp (y_1^\sharp, y_2^\sharp) \triangleq x_1^\sharp \sqsubseteq_1^\sharp y_1^\sharp \wedge x_2^\sharp \sqsubseteq_2^\sharp y_2^\sharp$$

and the concretization:

$$\gamma(x_1^\sharp, x_2^\sharp) \triangleq \gamma_1(x_1^\sharp) \sqcap^\flat \gamma_2(x_2^\sharp)$$

Let f_1^\sharp be an abstract transformer in $(\mathbb{D}_1^\sharp, \sqsubseteq_1^\sharp)$ and f_2^\sharp be an abstract transformer in $(\mathbb{D}_2^\sharp, \sqsubseteq_2^\sharp)$, both f_1^\sharp and f_2^\sharp are sound abstractions of concrete transformer f^\flat in $(\mathbb{D}^\flat, \sqsubseteq^\flat)$, we can then define the *product transformer*:

$$f^\sharp(x_1^\sharp, x_2^\sharp) \triangleq (f_1^\sharp(x_1^\sharp), f_2^\sharp(x_2^\sharp))$$

which is a sound abstraction of f^\flat . Comparing to compute each component separately, f^\sharp does actually not bring any precision improvement. This can be corrected by adding a reduction step that propagates information: let $\rho^\sharp \in \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$ be a function satisfying both the soundness condition $(\gamma \circ \rho^\sharp)(x^\sharp) = \gamma(x^\sharp)$ and the improvement condition $\rho^\sharp(x^\sharp) \sqsubseteq^\sharp x^\sharp$, then we calculate $\rho^\sharp \circ f^\sharp$ instead of f^\sharp . ρ^\sharp is called *reduction* which is a lower closure operator in \mathbb{D}^\sharp . $\rho^\sharp \circ f^\sharp$ may be not as precise as the best abstract transformer for f^\flat in \mathbb{D}^\sharp , But it is more precise than f^\sharp or at least has the same precision of f^\sharp . When both α_1 and α_2 exist in, respectively, $(\mathbb{D}_1^\sharp, \sqsubseteq_1^\sharp)$ and $(\mathbb{D}_2^\sharp, \sqsubseteq_2^\sharp)$, an optimal reduction can be defined as $\rho^\sharp(x^\sharp) \triangleq ((\alpha_1 \circ \gamma)(x^\sharp), (\alpha_2 \circ \gamma)(x^\sharp))$. Sometimes, ρ^\sharp is not easily computable due to lack of efficient algorithm, or does not exist due to lack of proper abstraction functions. We can still propagate information by using so-called *partial reduction*. More details can be

found in [CCF⁺07].

If one or both abstract domain does not satisfy the ascending chain condition, so does the product domain $\mathbb{D}_1^\sharp \times \mathbb{D}_2^\sharp$ and we need to define a widening and/or narrowing. We can define the widening $\nabla^\sharp \in (\mathbb{D}_1^\sharp \times \mathbb{D}_2^\sharp) \rightarrow (\mathbb{D}_1^\sharp \times \mathbb{D}_2^\sharp)$ component-wise as:

$$\nabla^\sharp \triangleq (\nabla_1^\sharp, \nabla_2^\sharp) \text{ or } (\nabla_1^\sharp, \sqcup_2^\sharp) \text{ or } (\sqcup_1^\sharp, \nabla_2^\sharp)$$

in case \mathbb{D}_1^\sharp or \mathbb{D}_2^\sharp satisfies the ascending chain condition and no widening is needed. A narrowing $\Delta^\sharp \in (\mathbb{D}_1^\sharp \times \mathbb{D}_2^\sharp) \rightarrow (\mathbb{D}_1^\sharp \times \mathbb{D}_2^\sharp)$ can be defined the same way as widening.

1.1.6 Abstract Domains

In abstract interpretation, all computations take place in the abstract domain which needs careful design. Normally, designing an abstract domain includes two aspects:

- Domain representation: choosing a computer-representable set of elements which is the over-approximation of the concrete semantics.
- Domain operators: designing a computable abstraction of each concrete function in the concrete domain \mathbb{D}^b . When the abstract domain does not satisfy the ascending chain condition and/or descending chain condition, a widening and/or narrowing operator(s) should also be included.

We then give the formal definition of abstract domain:

Definition 1.1.4 (Abstract Domain). An abstract domain can be defined as $(\mathbb{D}^\sharp, \sqsubseteq^\sharp, \alpha, \gamma, \perp^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp, \nabla^\sharp, \Delta^\sharp, f^\sharp, b^\sharp, \dots)$ where we have:

- a set \mathbb{D}^\sharp whose elements are computer-representable,
- a partial order \sqsubseteq^\sharp on \mathbb{D}^\sharp together with a algorithm to compare abstract elements,

- an abstraction function $\alpha : \mathbb{D}^b \rightarrow \mathbb{D}^\#$ and a concretization function $\gamma : \mathbb{D}^\# \rightarrow \mathbb{D}^b$,
- an infimum $\perp^\#$ and a supremum $\top^\#$,
- algorithms to compute sound abstractions $\sqcup^\#$ and $\sqcap^\#$ of, respectively, \sqcup^b and \sqcap^b ,
- a widening $\nabla^\#$ and/or an efficient narrowing $\Delta^\#$, if $\mathbb{D}^\#$ has strictly increasing infinite chains and/or strictly decreasing infinite chains,
- abstract transformers $f^\#, b^\#, \dots$ to compute sound abstractions of concrete semantics of concrete transformers.

□

Note that, best abstraction and best abstract transformers are preferred when they exist and are computable, but this is not mandatory. The cost of abstract operations and transformers should always be the first to be taken into consideration. Hence, every domain designer must make a reasonable tradeoff between efficiency and precision.

1.2 A Simple Programming Language

We present a minimal imperative programming language with its abstract syntax and formal semantics in the abstract interpretation framework that will be used in the following chapters. It's turing complete, easy to formalize the semantics and extendable. We also present an example of application of abstract interpretation to this simple programming language.

1.2.1 Abstract Syntax

The *syntax* of a programming language describes how to write programs. The *abstract syntax* describes the abstract syntax trees (AST) representing the syntactic structure of

source code.

Definition 1.2.1 (Abstract syntax).

$x \in \mathbb{X}$	variables
$E \in \mathbb{E} ::= 1 \mid x \mid ? \mid E_1 - E_2$	expressions
$B \in \mathbb{B} ::= E_1 < E_2 \mid E_1 \uparrow E_2$	boolean expressions
$C \in \mathbb{C} ::= \text{skip} \mid x = E \mid C_1 ; C_2 \mid$ $\text{if } (B) \{C_1\} \text{ else } \{C_2\} \mid$ $\text{while } (B) \{C\}$	commands
$A \in \mathbb{A} ::= \text{skip} \mid x = E \mid B \mid \neg B$	actions

□

1 is the only number. The other integers are abbreviations like $0 = 1 - 1$, $-1 = 0 - 1$, etc. $?$ returns random number which could be any integer value. \uparrow is “nand” i.e. “not and” logical operation. All other boolean operations can be defined in term of \uparrow such as $\neg B \triangleq B \uparrow B$, $B_1 \wedge B_2 \triangleq (B_1 \uparrow B_2) \uparrow (B_1 \uparrow B_2)$, etc. $<$ is the only comparison operation. All others can be defined in term of $<$ such as $E_1 \leq E_2 \triangleq \neg(E_2 < E_1)$, $E_1 == E_2 \triangleq E_1 \leq E_2 \wedge E_2 \leq E_1$, etc. Commands are similar to the C programming language except skip which is null/NOP execution step. Actions describe elementary indivisible program computation steps that is no operation skip, assignments $x = E$ executed by the program or conditions tested by the program. The action B records that the Boolean expression B evaluated to *true* (tt). The action $\neg B$ records that the Boolean expression B evaluated to *false* (ff).

1.2.2 States

States record the current values of variables in the environment/memory as well as a label/control point specifying what remains to be executed (nothing for stop). For example the state of the program $\text{while } (x \neq 0) \{ x = x + 1; \}$ just before executing the loop body is $\langle L, \rho \rangle$ where the set of variables is $\mathbb{X} = \{x\}$, the values $\mathcal{V} = \mathbb{Z}$ are integers, the environment $\rho \in \mathbb{X} \rightarrow \mathcal{V}$ maps x to its value $\rho(x)$, and $L \triangleq x = x + 1; \text{while } (x \neq 0) \{x = x + 1; \}$ denotes what remains to be done when execution is at L , i.e., execute the loop body and then repeat the loop entry.

Definition 1.2.2 (States).

$$\begin{array}{ll}
 L \in \mathbb{L} ::= C \mid \text{stop} & \text{labels} \\
 v \in \mathcal{V} & \text{values} \\
 \rho \in \mathcal{E} \triangleq \mathbb{X} \rightarrow \mathcal{V} & \text{environments} \\
 \sigma \in \Sigma \triangleq \mathbb{L} \times \mathcal{E} & \text{states}
 \end{array}$$

□

The environment assignment $\rho[x := v]$ of value v to the variable x in environment ρ is such that

$$\begin{array}{l}
 \rho[x := v](x) \triangleq v \\
 \rho[x := v](y) \triangleq \rho(y) \text{ when } x \neq y
 \end{array}$$

1.2.3 Traces

A trace π of length $|\pi| \triangleq 1$ is reduced to a single state $\pi = \sigma_0$. A trace π of length $|\pi| \triangleq n > 1$ is a pair $\pi = \langle \bar{\pi}, \underline{\pi} \rangle$ of a finite sequence $\bar{\pi} = \sigma_0 \sigma_1 \dots \sigma_{n-1} \in \Sigma^n$ of states separated by a finite sequence $\underline{\pi} = A_0 A_1 \dots A_{n-2} \in \mathbb{A}^{n-1}$ of actions, which we can write as $\sigma_0 \xrightarrow{A_0} \sigma_1 \xrightarrow{A_1} \dots \xrightarrow{A_{n-2}} \sigma_{n-1}$ and interpret as an observation of an execution that starts

from state σ_0 , such that in state σ_i , the execution of action A_i leads to next state σ_{i+1} , $i = 0, 1, \dots, n - 2$. Note that the first state σ_0 in a trace π may not be an initial state as well as the last one σ_{n-1} in the trace may not be the final state. This is because, in general, the observation of an execution may start at any time and stop at any later time during the execution.

Formally, Π^+ denotes the set of all finite traces while $\Pi^* \triangleq \{\varepsilon\} \cup \Pi^+$ also includes the empty trace ε corresponding to no observation. The set of all infinite traces is denoted Π^∞ so that the set of all traces is $\Pi \triangleq \Pi^* \cup \Pi^\infty$.

1.2.4 Trace Semantics of a Simple Programming Language

The semantics describes all possible observations of executions for all programs of the language. Hence, the semantics of a programming language can be formalized by traces.

Let the semantics $\mathcal{S}^t[[C]]$ of a command C be a set of traces $\pi \in \Pi$ which can be finite $\pi \in \mathcal{S}^{t^*}[[C]]$, where the execution terminates, or infinite $\pi \in \mathcal{S}^{t^\infty}[[C]]$, where the execution does not terminate. So we have $\mathcal{S}^t[[C]] \triangleq \mathcal{S}^{t^*}[[C]] \cup \mathcal{S}^{t^\infty}[[C]]$.

We define a set of infinite traces as the limits of a set \mathcal{S} of finite traces where the set of infinite traces have all their non-empty finite prefixes in set \mathcal{S} of finite traces.

Definition 1.2.3 (Infinite limits of a set of finite traces).

$$\begin{aligned} \lim^t &\in \wp(\Pi^*) \rightarrow \wp(\Pi^\infty) \\ \lim^t \mathcal{S} &\triangleq \{ \pi \in \Pi^\infty \mid \forall \pi_1 \in \Pi^* : (\exists \pi_2 \in \Pi^\infty : \pi = \pi_1 \xrightarrow{A} \pi_2) \implies \\ &\quad (\exists \pi_3 \in \Pi^* : \exists \pi_4 \in \Pi^\infty : \pi = \pi_1 \xrightarrow{A} \pi_3 \xrightarrow{A'} \pi_4 \wedge \pi_1 \xrightarrow{A} \pi_3 \in \mathcal{S}) \} \end{aligned}$$

□

We then define the trace semantics $\mathcal{S}^t[[C]]$ of commands C by induction on the syntactic structure of the commands.

stop When starting, the execution of the stop command finishes immediately. It does not have possible non-terminating behavior.

Definition 1.2.4 (Trace semantics of the stop command).

$$\begin{aligned}\mathcal{S}^{t*}[\text{stop}] &\triangleq \{\langle \text{stop}, \rho \rangle \mid \rho \in \mathcal{E}\} \\ \mathcal{S}^{t\infty}[\text{stop}] &\triangleq \emptyset\end{aligned}$$

□

skip The execution of the skip command is just one step that does not change the value of any variable. Same as stop, it does not have possible non-terminating behavior.

Definition 1.2.5 (Trace semantics of the skip command).

$$\begin{aligned}\mathcal{S}^{t*}[\text{skip}] &\triangleq \{\langle \text{skip}, \rho \rangle \xrightarrow{\text{skip}} \langle \text{stop}, \rho \rangle \mid \rho \in \mathcal{E}\} \\ \mathcal{S}^{t\infty}[\text{skip}] &\triangleq \emptyset\end{aligned}$$

□

Assignment The execution of the assignment command $x = E$ is just one step that changes the value of variable x by assigning any one of the possible values v of the expression E in the current environment/memory state ρ . Same as stop and skip, it does not have possible non-terminating behavior.

Definition 1.2.6 (Trace semantics of the assignment command).

$$\begin{aligned}\mathcal{S}^{t*}[x = E] &\triangleq \{\langle x = E, \rho \rangle \xrightarrow{x=E} \langle \text{stop}, \rho[x := v] \rangle \mid \rho \in \mathcal{E} \wedge v \in \mathcal{E}[\![E]\!]\rho\} \\ \mathcal{S}^{t\infty}[x = E] &\triangleq \emptyset\end{aligned}$$

□

Conditional A terminating (resp. non-terminating) execution of a conditional command $\text{if (B) } \{C_1\} \text{ else } \{C_2\}$ consists of an observation of the test, with action B when evaluation of the test in the current environment/memory state ρ is *true*, followed by a terminating (rest. non-terminating) execution of C_1 , or action $\neg B$ when evaluation of the test in the current environment/memory state ρ is *false*, followed by a terminating (rest. non-terminating) execution of C_2 .

Definition 1.2.7 (Trace semantics of the conditional command).

$$\begin{aligned}
\mathcal{S}^{t*}[\text{if (B) } \{C_1\} \text{ else } \{C_2\}] &\triangleq \\
&\{ \langle \text{if (B) } \{C_1\} \text{ else } \{C_2\}, \rho \rangle \xrightarrow{B} \langle C_1, \rho \rangle \xrightarrow{A} \pi \mid \\
&\quad \rho \in \mathcal{E} \wedge \text{true} \in \mathcal{E}[\![B]\!] \rho \wedge \langle C_1, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^{t*}[\![C_1]\!]\} \\
\cup &\{ \langle \text{if (B) } \{C_1\} \text{ else } \{C_2\}, \rho \rangle \xrightarrow{\neg B} \langle C_2, \rho \rangle \xrightarrow{A} \pi \mid \\
&\quad \rho \in \mathcal{E} \wedge \text{false} \in \mathcal{E}[\![B]\!] \rho \wedge \langle C_2, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^{t*}[\![C_2]\!]\} \\
\mathcal{S}^{t\infty}[\text{if (B) } \{C_1\} \text{ else } \{C_2\}] &\triangleq \\
&\{ \langle \text{if (B) } \{C_1\} \text{ else } \{C_2\}, \rho \rangle \xrightarrow{B} \langle C_1, \rho \rangle \xrightarrow{A} \pi \mid \\
&\quad \rho \in \mathcal{E} \wedge \text{true} \in \mathcal{E}[\![B]\!] \rho \wedge \langle C_1, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^{t\infty}[\![C_1]\!]\} \\
\cup &\{ \langle \text{if (B) } \{C_1\} \text{ else } \{C_2\}, \rho \rangle \xrightarrow{\neg B} \langle C_2, \rho \rangle \xrightarrow{A} \pi \mid \\
&\quad \rho \in \mathcal{E} \wedge \text{false} \in \mathcal{E}[\![B]\!] \rho \wedge \langle C_2, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^{t\infty}[\![C_2]\!]\}
\end{aligned}$$

□

Sequence The terminating execution of a command sequence $C_1 ; C_2$ is a terminating execution of C_1 followed by a terminating execution of C_2 . The non-terminating execution of a command sequence $C_1 ; C_2$ is either a non-terminating execution of C_1 or a terminating execution of C_1 followed by a non-terminating execution of C_2 . We first define trace sequencing $\pi ; C$ to be the trace π modified such that C is appended to all control states of the trace π . So if π describes some computation then $\pi ; C$ describes this same

computation to be following by some execution of C.

Definition 1.2.8 (Trace semantics of the command sequence).

$$\begin{aligned}
\mathcal{S}^{t*}[[C_1; C_2]] &\triangleq \{(\pi \ ; C_2) \xrightarrow{A} \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \mid \rho \in \mathcal{E} \wedge \\
&\quad \pi \xrightarrow{A} \langle \text{stop}, \rho \rangle \in \mathcal{S}^{t*}[[C_1]] \wedge \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \in \mathcal{S}^{t*}[[C_2]]\} \\
\mathcal{S}^{t\infty}[[C_1; C_2]] &\triangleq \{(\pi \ ; C_2) \mid \pi \in \mathcal{S}^{t\infty}[[C_1]]\} \\
&\cup \{(\pi \ ; C_2) \xrightarrow{A} \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \mid \rho \in \mathcal{E} \wedge \\
&\quad \pi \xrightarrow{A} \langle \text{stop}, \rho \rangle \in \mathcal{S}^{t*}[[C_1]] \wedge \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \in \mathcal{S}^{t\infty}[[C_2]]\}
\end{aligned}$$

□

Loop The trace semantics of a loop can be defined in the least fixpoint form. This consists in defining a set of trace transformer $\mathcal{F}^{ti}[\text{while}(B) \{C\}]X$ that introduces base cases and extends already observed iterates in X by one more iteration taking its fixpoint. The terminating executions of a loop are those that iterate 0 or more times where, at each time the loop body terminates as long as the loop condition B is *true* and terminate as soon as the loop condition B becomes *false*. The non-terminating executions of a loop can be defined as the limit of the prefix traces corresponding to finite iterations in the loop where, at each time the loop body terminates plus those execution where after a finite number of iterations, the loop body does not terminate (which can happen with nested loops).

Definition 1.2.9 (Trace semantics of while loop).

$$\begin{aligned}
\mathcal{F}^{ti}[\text{while}(B) \{C\}]X &\triangleq \{\langle \text{while}(B) \{C\}, \rho \rangle \mid \rho \in \mathcal{E}\} \cup \\
&\{\pi \xrightarrow{A} \langle \text{while}(B) \{C\}, \rho \rangle \xrightarrow{B} (\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \ ; \ \text{while}(B) \{C\} \mid \\
&\quad \pi, \pi' \in \Pi^* \wedge \pi \xrightarrow{A} \langle \text{while}(B) \{C\}, \rho \rangle \in X \wedge \text{true} \in \mathcal{E}[[B]]\rho \wedge \\
&\quad (\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \in \mathcal{S}^{t*}[[C]]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}^{ti}[\text{while (B) } \{C\}] &\triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while (B) } \{C\}] \\
\mathcal{S}^{t*}[\text{while (B) } \{C\}] &\triangleq \{ \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \xrightarrow{\neg B} \langle \text{stop}, \rho \rangle \mid \\
&\quad \pi \in \Pi^* \wedge \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \in \mathcal{S}^{ti}[\text{while (B) } \{C\}] \wedge \text{false} \in \mathcal{E}[\text{B}]\rho \} \\
\mathcal{S}^{t\infty}[\text{while (B) } \{C\}] &\triangleq \\
&\lim^t \{ \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \in \mathcal{S}^{ti}[\text{while (B) } \{C\}] \mid \text{true} \in \mathcal{E}[\text{B}]\rho \} \cup \\
&\{ \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \xrightarrow{B} \langle C; \text{while (B) } \{C\}, \rho \rangle \xrightarrow{A'} \pi' \ ; \ \text{while (B) } \{C\} \mid \\
&\quad \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \in \mathcal{S}^{ti}[\text{while (B) } \{C\}] \wedge \text{true} \in \mathcal{E}[\text{B}]\rho \wedge \\
&\quad \langle C, \rho \rangle \xrightarrow{A'} \pi' \in \mathcal{S}^{t\infty}[C] \}
\end{aligned}$$

□

1.2.5 Action Path Semantics: An Example of Abstract Interpretation

In program analysis, especially in data-flow analysis [Hec77], it is common to represent programs by control flow graphs (or flowcharts, or some equivalent intermediate representation) and then to reason on sequences of actions along all paths in these control flow graphs. We will define the action path semantics $\mathcal{G}^a[\mathbb{G}[C]]$ of the control flow graph $\mathbb{G}[C]$ of command C as an abstract interpretation α^a of the trace semantics $\mathcal{S}^t[C]$.

Action Path Abstraction

The action path abstraction $\alpha^a(\mathcal{S})$ collects the set of action paths, which are sequences of actions performed along the traces of a trace semantics \mathcal{S} and ignores anything about states.

We denote \mathbb{A}^+ the set of non-empty finite action paths, \mathbb{A}^* the set of finite, possibly empty, action paths so that $\mathbb{A}^* \triangleq \{\varepsilon\} \cup \mathbb{A}^+$ and \mathbb{A}^∞ the set of infinite action paths.

Given a trace $\pi = \langle \bar{\pi}, \underline{\pi} \rangle$, $\alpha^a(\pi) \triangleq \underline{\pi}$ collects the sequence of actions $\underline{\pi}$ executed along

that trace, which may be empty ε for traces reduced to a single state. It follows that $\alpha^a(\pi \xrightarrow{A} \pi') = \alpha^a(\pi) \cdot A \cdot \alpha^a(\pi')$ and $\alpha^a(\pi \text{ ; } C) = \alpha^a(\pi)$.

Definition 1.2.10 (Action path abstraction). Given a set of traces \mathcal{S} ,

$$\begin{aligned} \alpha^a &\in \wp(\Pi) \rightarrow \wp(\mathbb{A}^* \cup \mathbb{A}^\infty) \\ \alpha^a(\mathcal{S}) &\triangleq \{\alpha^a(\pi) \mid \pi \in \mathcal{S}\} \end{aligned}$$

collects the sequences of actions executed along the traces of \mathcal{S} . □

Note that α^a preserves both unions and intersections, and we have the following theorem:

Theorem 1.2.1 (Homomorphic Abstraction). *Given a function $h : C \mapsto A$, let $\alpha_h(X) = \{h(x) \mid x \in X\}$ and $\gamma_h(Y) = \{x \mid h(x) \in Y\}$, then α_h and γ_h form a Galois connection:*

$$(\wp(C), \subseteq) \xrightleftharpoons[\alpha_h]{\gamma_h} (\wp(A), \subseteq) \tag{1.1}$$

Proof. For all $X \in \wp(C)$ and $Y \in \wp(A)$,

$$\begin{aligned} &\alpha_h(X) \subseteq Y \\ \iff &\{h(x) \mid x \in X\} \subseteq Y && \text{\{definition of } \alpha_h\}} \\ \iff &\forall x \in X : h(x) \in Y && \text{\{definition of } \subseteq\}} \\ \iff &X \subseteq \{x \mid h(x) \in Y\} && \text{\{definition of } \subseteq\}} \\ \iff &X \subseteq \gamma_h(Y) && \text{\{definition of } \gamma_h\}} \end{aligned}$$

□

Hence, by defining $\gamma^a(\mathcal{A}) \triangleq \{\pi \mid \alpha^a(\pi) \in \mathcal{A}\}$, we will have

Corollary 1.2.2.

$$(\wp(\Pi), \subseteq) \xrightleftharpoons[\alpha^a]{\gamma^a} (\wp(\mathbb{A}^* \cup \mathbb{A}^\infty), \subseteq)$$

Proof. By Theorem 1.2.1 where h is α^a . □

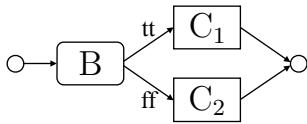
Control Flow Graph

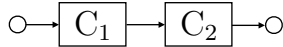
A *control flow graph* (V, E) of a program is usually a directed graph, in which nodes correspond to the actions in the program and the edges represent the possible flow of control and might be weighted with tt and ff. Normally, a CFG (short for control flow graph) has a single point of entry which is denoted entry, and a single point of exit which is denoted exit.

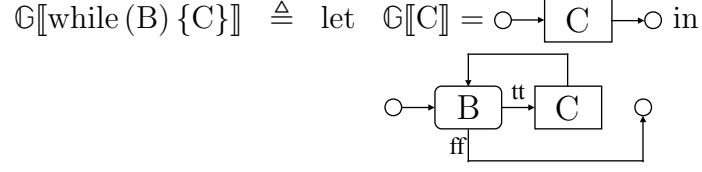
The CFGs can be defined in an inductive manner. Let $\mathbb{G}[[C]]$ be a control flow graph of a command C , we then build it by the structural induction on the syntax of the command C :

$$\mathbb{G}[[\text{skip}]] \triangleq \text{○} \rightarrow \boxed{\text{skip}} \rightarrow \text{○}$$

$$\mathbb{G}[[x := E]] \triangleq \text{○} \rightarrow \boxed{x := E} \rightarrow \text{○}$$

$$\mathbb{G}[[\text{if } (B) \{C_1\} \text{ else } \{C_2\}]] \triangleq \text{let } \begin{array}{l} \mathbb{G}[[C_1]] = \text{○} \rightarrow \boxed{C_1} \rightarrow \text{○} \text{ and} \\ \mathbb{G}[[C_2]] = \text{○} \rightarrow \boxed{C_2} \rightarrow \text{○} \text{ in} \end{array}$$


$$\mathbb{G}[[C_1; C_2]] \triangleq \text{let } \begin{array}{l} \mathbb{G}[[C_1]] = \text{○} \rightarrow \boxed{C_1} \rightarrow \text{○} \text{ and} \\ \mathbb{G}[[C_2]] = \text{○} \rightarrow \boxed{C_2} \rightarrow \text{○} \text{ in} \end{array}$$




Action Path Semantics of CFGs

Following the definition of trace semantics $\mathcal{S}^t[\text{C}]$ of command C, we define the action path semantics $\mathcal{G}^a[\mathbb{G}[\text{C}]]$ of CFG $\mathbb{G}[\text{C}]$ of command C as the union of finite sequences of actions $\mathcal{G}^{a*}[\mathbb{G}[\text{C}]]$ abstracting the finite trace semantics $\mathcal{S}^{t*}[\text{C}]$ of the command and infinite sequences of actions $\mathcal{G}^{a\infty}[\mathbb{G}[\text{C}]]$ abstracting the infinite trace semantics $\mathcal{S}^{t\infty}[\text{C}]$ of the command, i.e., $\mathcal{G}^a[\mathbb{G}[\text{C}]] \triangleq \mathcal{G}^{a*}[\mathbb{G}[\text{C}]] \cup \mathcal{G}^{a\infty}[\mathbb{G}[\text{C}]]$.

We first define the limits of sets of action paths which is similar to the limits of sets of finite traces in definition 1.2.3.

Definition 1.2.11 (Infinite limits of a set of finite action paths).

$$\begin{aligned} \lim^a &\in \wp(\mathbb{A}^*) \rightarrow \wp(\mathbb{A}^\infty) \\ \lim^a \mathcal{S} &\triangleq \{ \omega \in \mathbb{A}^\infty \mid \forall \omega_1 \in \mathbb{A}^* : (\exists \omega_2 \in \mathbb{A}^\infty : \omega = \omega_1 \cdot \omega_2) \implies \\ &\quad (\exists \omega_3 \in \mathbb{A}^* : \exists \omega_4 \in \mathbb{A}^\infty : \omega = \omega_1 \cdot \omega_3 \cdot \omega_4 \wedge \omega_1 \cdot \omega_3 \in \mathcal{S}) \} \end{aligned}$$

□

Hence,

Definition 1.2.12 (Action path semantics of CFGs).

$$\mathcal{G}^{a*}[\circ \rightarrow \boxed{\text{skip}} \rightarrow \circ] \triangleq \{\text{skip}\}$$

$$\mathcal{G}^{a\infty}[\circ \rightarrow \boxed{\text{skip}} \rightarrow \circ] \triangleq \emptyset$$

$$\mathcal{G}^{a*}[\circ \rightarrow \boxed{x := E} \rightarrow \circ] \triangleq \{x = E\}$$

$$\mathcal{G}^{a\infty}[\circ \rightarrow \boxed{x := E} \rightarrow \circ] \triangleq \emptyset$$

$$\begin{aligned} \mathcal{G}^{a*}[\circ \rightarrow \boxed{B} \begin{array}{l} \xrightarrow{\text{tt}} \boxed{C_1} \\ \xrightarrow{\text{ff}} \boxed{C_2} \end{array} \rightarrow \circ] &\triangleq \{B\} \cdot \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_1} \rightarrow \circ] \\ &\cup \{\neg B\} \cdot \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_2} \rightarrow \circ] \end{aligned}$$

$$\begin{aligned} \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{B} \begin{array}{l} \xrightarrow{\text{tt}} \boxed{C_1} \\ \xrightarrow{\text{ff}} \boxed{C_2} \end{array} \rightarrow \circ] &\triangleq \{B\} \cdot \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C_1} \rightarrow \circ] \\ &\cup \{\neg B\} \cdot \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C_2} \rightarrow \circ] \end{aligned}$$

$$\mathcal{G}^{a*}[\circ \rightarrow \boxed{C_1} \rightarrow \boxed{C_2} \rightarrow \circ] \triangleq \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_1} \rightarrow \circ] \cdot \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_2} \rightarrow \circ]$$

$$\begin{aligned} \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C_1} \rightarrow \boxed{C_2} \rightarrow \circ] &\triangleq \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C_1} \rightarrow \circ] \\ &\cup \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_1} \rightarrow \circ] \cdot \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C_2} \rightarrow \circ] \end{aligned}$$

$$\mathcal{F}^{ai}[\circ \rightarrow \boxed{B} \begin{array}{l} \xrightarrow{\text{tt}} \boxed{C} \\ \xrightarrow{\text{ff}} \boxed{B} \end{array} \rightarrow \circ] X \triangleq \{\varepsilon\} \cup X \cdot \{B\} \cdot \mathcal{G}^{a*}[\circ \rightarrow \boxed{C} \rightarrow \circ]$$

$$\mathcal{G}^{a*}[\circ \rightarrow \boxed{B} \begin{array}{l} \xrightarrow{\text{tt}} \boxed{C} \\ \xrightarrow{\text{ff}} \boxed{B} \end{array} \rightarrow \circ] \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ai}[\circ \rightarrow \boxed{B} \begin{array}{l} \xrightarrow{\text{tt}} \boxed{C} \\ \xrightarrow{\text{ff}} \boxed{B} \end{array} \rightarrow \circ] \cdot \{\neg B\}$$

$$\begin{aligned} \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{B} \begin{array}{l} \xrightarrow{\text{tt}} \boxed{C} \\ \xrightarrow{\text{ff}} \boxed{B} \end{array} \rightarrow \circ] &\triangleq \text{let } W = \text{lfp}^{\subseteq} \mathcal{F}^{ai}[\circ \rightarrow \boxed{B} \begin{array}{l} \xrightarrow{\text{tt}} \boxed{C} \\ \xrightarrow{\text{ff}} \boxed{B} \end{array} \rightarrow \circ] \text{ in} \\ &W \cdot \{B\} \cdot \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C} \rightarrow \circ] \cup \text{lim}^a W \end{aligned}$$

□

Informally, using regular expressions, we have:

$$\begin{aligned}
\mathcal{G}^{a^*} \llbracket \circ \rightarrow \boxed{B} \xrightarrow{tt} \boxed{C} \rightarrow \circ \rrbracket &\triangleq (\{B\} \cdot \mathcal{G}^{a^*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^* \cdot \{\neg B\} \\
\mathcal{G}^{a^\infty} \llbracket \circ \rightarrow \boxed{B} \xrightarrow{tt} \boxed{C} \rightarrow \circ \rrbracket &\triangleq (\{B\} \cdot \mathcal{G}^{a^*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^* \cdot \{B\} \cdot \mathcal{G}^{a^\infty} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket \\
&| (\{B\} \cdot \mathcal{G}^{a^*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^\infty
\end{aligned}$$

Ideally, we would like $\mathcal{G}^a \llbracket \mathbb{G}[C] \rrbracket = \alpha^a(\mathcal{S}^t \llbracket C \rrbracket)$. Unfortunately, this is impossible due to the lack of environment/memory information during execution in the action path semantics. The following theorem states that the action path semantics is an over-approximation of the action paths that would be collected directly from the trace semantics.

Theorem 1.2.3. $\alpha^a(\mathcal{S}^t \llbracket C \rrbracket) \subseteq \mathcal{G}^a \llbracket \mathbb{G}[C] \rrbracket$.

Proof. See appendix. □

The above theorem is a good example of approximate abstraction where the abstraction of the concrete semantics must be over-approximated in the abstract since it cannot be exactly calculated in the abstract only.

1.3 Numerical Static Analysis

In this section, we discuss the static analysis that automatically discovers the numerical properties of program variable by using so-called *numerical abstract domains*. We then present two classical numerical abstract domains - interval and polyhedra.

1.3.1 Discovering properties of numerical variables

Given a program, the aim of numerical static analysis is to find out the possible values of all variables at each program point. This can be done by observing the set of *reachable states* $\mathcal{R} \subseteq \Sigma$, which is an abstraction of trace semantics \mathcal{S} . The reachable state abstraction function can be defined as:

Definition 1.3.1 (Reachable state abstraction). Given a set of traces \mathcal{S} ,

$$\begin{aligned} \alpha^r &\in \wp(\Pi) \rightarrow \wp(\Sigma) \\ \alpha^r(\mathcal{S}) &\triangleq \{\sigma \mid \exists \sigma_0 \xrightarrow{A_0} \dots \sigma_n \in \mathcal{S} : \exists i \leq n : \sigma = \sigma_i\} \end{aligned}$$

□

We model program semantics as a *labelled transition system* $(\Sigma, \mathcal{I}, \mathbb{A}, \tau)$ where:

- Σ : a set of states;
- $\mathcal{I} \subseteq \Sigma$: a set of initial states;
- \mathbb{A} : a set of actions;
- $\tau \in \Sigma \times \mathbb{A} \times \Sigma$: a transition relation.

Let \top be all possible values of all variables in the environment/memory and $\perp = \emptyset$, we denote $\mathcal{I} = \{\langle L_0, \top \rangle \mid L_0 \in \mathbb{L}\} \cup \{\langle L_i, \perp \rangle \mid L_i \in \mathbb{L}, i \in \mathbb{N}, i \geq 1\}$.

The set of reachable states \mathcal{R} can be constructed by fixpoint. We define the function $r \in \wp(\Sigma) \rightarrow \wp(\Sigma)$ as:

$$r(X) \triangleq \mathcal{I} \cup \{\sigma \mid \exists \sigma' \in X : \exists A \in \mathbb{A} : (\sigma', A, \sigma) \in \tau\}$$

and we have $\mathcal{R} = \text{lfp}(r)$.

We can then lift the reachable state semantics $\wp(\mathbb{L} \times (\mathcal{X} \rightarrow \mathcal{V}))$ to $\mathbb{L} \rightarrow \wp(\mathcal{X} \rightarrow \mathcal{V})$ which is called *collecting semantics*. In this form, we can easily get the strongest properties of states at each program point by “collecting” all possible environments/memories. However, such environments/memories may often be infinite, or finite but too large to be represented in computer system. Hence, we need to seek further abstractions.

Let the concrete domain be $(\wp(\mathcal{X} \rightarrow \mathcal{V}), \subseteq, \cup, \cap, \top, \perp)$, our aim is to design an abstract domain with a computer-representable abstract version of concrete domain $\wp(\mathcal{X} \rightarrow \mathcal{V})$ and computable abstractions of any concrete semantic functions. In the past 40 years, a lot of numerical abstract domains had been introduced. In the remaining of this section, we recall two well-known numerical abstract domains: intervals and polyhedra.

1.3.2 Intervals

The *interval abstract domain* was first introduced in [CC76] by Cousot and Cousot. It uses an upper and a lower bound to abstract possible values for each variable. The interval abstract domain is very efficient and able to provide useful information on program executions. For example, it can be used to prove the absence of arithmetic overflow or out-of-bound array access.

Let \mathcal{Int} be the set of empty or non-empty intervals with bounds in \mathbb{Z} where

$$\mathcal{Int} \triangleq \{[a, b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \cup \{\perp_I\}$$

In particular, \mathcal{Int} contains $[-\infty, \infty]$ which represents the whole set \mathbb{Z} , and the singletons $[a, a]$ when $a \in \mathbb{Z}$. The empty interval is denoted by \perp_I . The partial order \sqsubseteq_I is defined as:

$$[a_1, b_1] \sqsubseteq_I [a_2, b_2] \triangleq a_1 \geq a_2 \wedge b_1 \leq b_2$$

and the abstraction and concretization functions are defined as:

$$\alpha_I(S) \triangleq \begin{cases} \perp_I & \text{if } S = \emptyset, \\ [\min S, \max S] & \text{otherwise.} \end{cases}$$

$$\gamma_I(\perp_I) \triangleq \emptyset;$$

$$\gamma_I([a, b]) \triangleq \{v \mid v \in \mathbb{Z} \wedge a \leq v \leq b\}.$$

The abstract meet \sqcap_I and join \sqcup_I are defined as:

$$[a, b] \sqcap_I [c, d] \triangleq [\max\{a, c\}, \min\{b, d\}];$$

$$[a, b] \sqcup_I [c, d] \triangleq [\min\{a, c\}, \max\{b, d\}].$$

Note that the abstract join \sqcup_I is not exact, e.g., $[1, 2] \sqcup_I [4, 5] = [1, 5]$ which contains spurious value 3. Then the widening ∇_I is defined as:

$$[a, b] \nabla_I [c, d] \triangleq \left[\begin{array}{l} \left\{ \begin{array}{ll} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{array} \right\}, \left\{ \begin{array}{ll} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{array} \right\} \end{array} \right]$$

The widening ∇_I is similar to \sqcup_I but ensures termination by replacing unstable upper bounds with $+\infty$ and lower bounds with $-\infty$, so that intervals cannot grow indefinitely.

At the end, we define several abstract expression evaluation functions as:

$$\begin{aligned}
-I[a, b] &\triangleq [-b, -a] \\
[a, b] +_I [c, d] &\triangleq [a + c, b + d] \\
[a, b] -_I [c, d] &\triangleq [a - d, b - c] \\
[a, b] \times_I [c, d] &\triangleq [\min\{ac, bc, ad, bd\}, \max\{ac, bc, ad, bd\}] \\
[a, b] /_I [c, d] &\triangleq \begin{cases} \perp_I & \text{if } c = d = 0 \\ [\min\{a/c, a/d, b/c, b/d\}, \\ \max\{a/c, a/d, b/c, b/d\}] & \text{else if } c \geq 0 \\ [-b, -a] /_I [-d, -c] & \text{else if } d \leq 0 \\ [a, b] /_I [c, 0] \sqcup_I [a, b] /_I [0, d] & \text{otherwise} \end{cases}
\end{aligned}$$

Then the interval abstract domain can be defined as the point-wise extension over $\mathbb{X} \rightarrow \mathcal{Int}$:

$$\begin{aligned}
\mathbb{D}^\# &\triangleq (\mathbb{X} \rightarrow \mathcal{Int}) \cup \{\perp^\#\} \\
R_1^\# \sqsubseteq^\# R_2^\# &\triangleq \forall x : R_1^\#(x) \sqsubseteq_I R_2^\#(x) \\
\alpha(R) &\triangleq \lambda x \in \mathbb{X} . \alpha_I(\{\rho(x) \mid \rho \in R\}) \\
R_1^\# \text{ op}^\# R_2^\# &\triangleq \lambda x \in \mathbb{X} . R_1^\#(x) \text{ op}_I R_2^\#(x) \text{ where } \text{op} = \sqcup, \sqcap, \nabla, +, -, \times, /.
\end{aligned}$$

Our description of interval abstract domain uses integer numbers \mathbb{Z} to represent bounds. It can also use real numbers \mathbb{R} . Note that the set of real numbers does not have effective computer representation and algorithms. Hence, in practice, computer representable rational numbers are commonly used which leads to a slightly weaker domain.

1.3.3 Polyhedra

The *polyhedra abstract domain* was first introduced in [CH78] by Cousot and Halbwachs. It uses a set of affine inequalities to abstract possible values for program variables. Compared to the interval abstract domain, polyhedra abstract domain is much more expressive and precise, but much less efficient of course. Another difference between the interval abstract domain and the polyhedra abstract domain is that the interval abstract domain abstracts each variable independently so it is not able to discover relationships between variables while the polyhedra abstract domain is able to. Those abstract domains who abstract each variable independently, such as the interval abstract domain, are called *non-relational abstract domains* while the other abstract domains who can discover relationships between variables, such as the polyhedra abstract domain, are called *relational abstract domains*.

The polyhedra abstract domain is based on the theory of linear algebra. It uses a set of (or conjunction of) linear inequalities to represent the property of a set of variables (x_1, \dots, x_n) . Normally, we need the polyhedra abstract domain to be closed, convex (and possibly unbounded) on \mathbb{Z}^n . In practice, there exists two representations for polyhedra:

- a finite set of *constraints* $\mathcal{C} = \{\sum_{i=1}^n a_{1i}x_i \leq b_1, \dots, \sum_{i=1}^n a_{mi}x_i \leq b_m\}$ which we usually denote as a pair $\langle A, \vec{B} \rangle$ where $A \in \mathbb{Z}^{m \times n}$ is a matrix and $\vec{B} \in \mathbb{Z}^m$ is a vector;
- a finite set of *generators*, that is, a set of *points* $P = \{\vec{P}_1, \dots, \vec{P}_p\}$ and a set of *rays* $R = \{\vec{R}_1, \dots, \vec{R}_r\}$ which we usually denote as a pair $\langle P, R \rangle$.

The concretization functions for both representations are defined as:

$$\begin{aligned} \gamma(\mathcal{C}) &\triangleq \gamma(\langle A, \vec{B} \rangle) \triangleq \{\vec{X} \mid A \times \vec{X} \leq \vec{B}\} \\ \gamma(\langle P, R \rangle) &\triangleq \{(\sum_{i=1}^p \lambda_i \vec{P}_i + \sum_{i=1}^r \mu_i \vec{R}_i) \mid \lambda_i \geq 0, \mu_i \geq 0, \sum_{i=1}^p \lambda_i = 1\} \end{aligned}$$

There is no abstraction function because some vector sets do not have a best over-

approximation as a closed convex polyhedron, e.g., $x^2 + y^2 \leq 1$.

Some of typical abstract operators in the polyhedra abstract domain are defined as:

$$\begin{aligned}
\langle P, R \rangle \sqsubseteq^\# \langle A, \vec{B} \rangle &\triangleq \forall i : A \times \vec{P}_i \leq \vec{B} \wedge \forall i : A \times \vec{R}_i \leq 0 \\
\langle P_1, R_2 \rangle \sqcup^\# \langle P_2, R_2 \rangle &\triangleq \langle P_1 \cup P_2, R_1 \cup R_2 \rangle \\
\mathcal{C}_1 \nabla^\# \mathcal{C}_2 &\triangleq \{c_1 \in \mathcal{C}_1 \mid \mathcal{C}_2 \sqsubseteq^\# \{c_1\}\} \cup \\
&\quad \{c_2 \in \mathcal{C}_2 \mid \exists c_1 \in \mathcal{C}_1 : \mathcal{C}_1 =^\# (\mathcal{C}_1 \setminus \{c_1\}) \cup \{c_2\}\}
\end{aligned}$$

The above widening is called standard widening for polyhedra. More details can be found in [BHRZ05]. Note that some of operations we perform on closed convex polyhedra are easy when those polyhedra are represented by a set of constraints (e.g., widening $\nabla^\#$) while some other operations are easy when those polyhedra are represented by a set of generators (e.g., abstract join $\sqcup^\#$). Moreover, some operations we perform on closed convex polyhedra are more simple when we use both representations (e.g., abstract ordering $\sqsubseteq^\#$). Hence, it is usually to use both representations in the polyhedra abstract domains thus a conversion algorithm is also included. A standard conversion algorithm is due to Chernikova and later improved by LeVerge [Ver94].

Chapter 2

Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) solving [BSST09] is concerned with the satisfiability of first-order formulas with respect to some background theories. In this chapter, we first recall the notion of first-order logic. We then recall the SMT problem. We introduce the notion of satisfiability and validity modulo theories, and recall the theories of interest in SMT. We also present the abstract DPLL(\mathcal{T}) algorithm for solving SMT problems, and Nelson-Oppen method for combining theories. Finally, we introduce SMT-LIB, a library designed for the SMT problem. The material introduced in this chapter serves as reference for the following chapters in this thesis.

2.1 First-Order Logic

In this section, we introduce first-order logic, which includes its syntax, interpretations, theories and models.

2.1.1 Syntax

The set $\mathbb{F}(\mathcal{X}, \mathbb{f}, \mathbb{p})$ of first-order formulas over a countable set of variables \mathcal{X} and a disjoint signature $\Sigma = \langle \mathbb{f}, \mathbb{p} \rangle$ (where \mathbb{f} is a set of function symbols and \mathbb{p} is a set of predicate symbols such that \mathbb{f} and \mathbb{p} are disjoint, i.e., $\mathbb{f} \cap \mathbb{p} = \emptyset$) is defined as:

Definition 2.1.1 (First-order logic).

x, y, z, \dots	$\in \mathcal{X}$	variables
a, b, c, \dots	$\in \mathbb{f}^0$	constants
f, g, h, \dots	$\in \mathbb{f}^n$	functions with arity $n \geq 1$
\mathbb{f}	$\triangleq \bigcup_{n \geq 0} \mathbb{f}^n$	
t	$\in \mathbb{T}(\mathcal{X}, \mathbb{f})$	terms
t	$::= x \mid c \mid f(t_1, t_2, \dots, t_n)$	
ff, tt	$\in \mathbb{p}^0$	propositions
p, q, r, \dots	$\in \mathbb{p}^n$	predicates with arity $n \geq 1$
\mathbb{p}	$\triangleq \bigcup_{n \geq 0} \mathbb{p}^n$	
a	$\in \mathbb{A}(\mathcal{X}, \mathbb{f}, \mathbb{p})$	atomic formulas
a	$::= \text{ff} \mid p(t_1, t_2, \dots, t_n) \mid \neg a$	
φ	$\in \mathbb{F}(\mathcal{X}, \mathbb{f}, \mathbb{p})$	quantified first-order formulas
φ	$::= a \mid \neg \varphi \mid \varphi \wedge \varphi \mid \exists x : \varphi$	

□

Note that all other boolean operators can be defined in term of \neg and \wedge such that $\varphi \vee \phi \triangleq \neg(\neg\varphi \wedge \neg\phi)$, $\varphi \Rightarrow \phi \triangleq \neg(\varphi \wedge \neg\phi)$, $\varphi \Leftrightarrow \phi \triangleq (\varphi \Rightarrow \phi) \wedge (\phi \Rightarrow \varphi)$. The universal quantifier \forall can be defined in term of the existential quantifier \exists and \neg such that $\forall x : \varphi \triangleq \neg(\exists x : \neg\varphi)$. A *literal* is an atomic formula or its negation. A *clause* is a disjunction $l_1 \vee l_2 \vee \dots \vee l_n$ of zero or more literals. A *CNF formula* is a conjunction

$c_1 \wedge c_2 \wedge \dots \wedge c_n$ of zero or more clauses. A first-order formula is *quantifier-free* if and only if it contains no quantifiers. In *first-order logic with equality*, there is a distinguished predicate $= (t_1, t_2)$ which we write $t_1 = t_2$.

2.1.2 Interpretations

Given a set $\mathbb{F}(x, f, p)$ of first-order formulas on a signature $\langle f, p \rangle$, an *interpretation* I is a couple $\langle I_{\mathcal{V}}, I_{\gamma} \rangle$ such that:

- $I_{\mathcal{V}}$ is a non-empty set of values,
- $\forall c \in \mathbb{F}^0 : I_{\gamma}(c) \in I_{\mathcal{V}}$,
- $\forall n \geq 1 : \forall f \in \mathbb{F}^n : I_{\gamma}(f) \in I_{\mathcal{V}}^n \rightarrow I_{\mathcal{V}}$,
- $\forall n \geq 0 : \forall p \in \mathbb{P}^n : I_{\gamma}(p) \in I_{\mathcal{V}}^n \rightarrow \mathcal{B}$

where $\mathcal{B} \triangleq \{false, true\} \subseteq I_{\mathcal{V}}$. Let the *environment* be a function from variables to the values in a given interpretation $I \in \mathfrak{S}$ such that:

$$\rho \in \mathcal{E} \triangleq x \rightarrow I_{\mathcal{V}}.$$

We say an interpretation I and an environment ρ satisfy a first-order formula φ , which is denoted by $I \models_{\rho} \varphi$, in the following way:

$$\begin{aligned} I \models_{\rho} a &\triangleq \llbracket a \rrbracket_I \rho \\ I \models_{\rho} \neg \varphi &\triangleq \neg(I \models_{\rho} \varphi) \\ I \models_{\rho} \varphi \wedge \phi &\triangleq (I \models_{\rho} \varphi) \wedge (I \models_{\rho} \phi) \\ I \models_{\rho} \exists x : \varphi &\triangleq \exists v \in I_{\mathcal{V}} : I \models_{\rho[x:=v]} \varphi \end{aligned}$$

where the *evaluation* $\llbracket a \rrbracket_{I\rho} \in \mathcal{B}$ of an atomic formula $a \in \mathbb{A}(\mathfrak{x}, \mathfrak{f}, \mathfrak{p})$ in the environment ρ is:

$$\begin{aligned}\llbracket \text{ff} \rrbracket_{I\rho} &\triangleq \text{false}, \\ \llbracket \neg a \rrbracket_{I\rho} &\triangleq \neg \llbracket a \rrbracket_{I\rho}, \\ \llbracket \mathfrak{p}(t_1, \dots, t_n) \rrbracket_{I\rho} &\triangleq I_\gamma(\mathfrak{p})(\llbracket t_1 \rrbracket_{I\rho}, \dots, \llbracket t_n \rrbracket_{I\rho}),\end{aligned}$$

where the evaluation $\llbracket t \rrbracket_{I\rho} \in I_\mathcal{V}$ of a term $t \in \mathbb{T}(\mathfrak{x}, \mathfrak{f})$ in the environment ρ is:

$$\begin{aligned}\llbracket \mathfrak{x} \rrbracket_{I\rho} &\triangleq \rho(\mathfrak{x}), \\ \llbracket \mathfrak{c} \rrbracket_{I\rho} &\triangleq I_\gamma(\mathfrak{c}), \\ \llbracket \mathfrak{f}(t_1, \dots, t_n) \rrbracket_{I\rho} &\triangleq I_\gamma(\mathfrak{f})(\llbracket t_1 \rrbracket_{I\rho}, \dots, \llbracket t_n \rrbracket_{I\rho}).\end{aligned}$$

In addition, we say an interpretation I and an environment ρ satisfy an equality in the first-order logic with equality such that:

$$I \models_\rho t_1 = t_2 \triangleq \llbracket t_1 \rrbracket_{I\rho} =_I \llbracket t_2 \rrbracket_{I\rho}$$

where $=_I$ is the unique reflexive, symmetric, antisymmetric, and transitive relation on $I_\mathcal{V}$.

2.1.3 Theories and Models

The set of *free variables* of a first-order formula φ can be defined inductively as the set of variables in the formula which are not in the scope of an existential quantifier. A *sentence* is a first-order formula with no free variables. A theory is a set of sentences [CK90] (which are called the theorems of the theory) under the same signature which should contain at least all the predicates and function symbols that appear in the theorems. The *language* of a theory is the set of quantified first-order formulas that contain no predicate or function symbol outside of the signature of the theory.

An Interpretation $I \in \mathfrak{S}$ is said to be a *model* of a first-order formula φ when

$$\exists \rho : I \models_{\rho} \varphi.$$

An interpretation $I \in \mathfrak{S}$ is said to be a *model* of a theory if and only if it is a model of all the theorems of the theory. We use

$$\mathcal{M}(\mathcal{T}) \triangleq \{I \in \mathfrak{S} \mid \forall \varphi \in \mathcal{T} : \exists \rho : I \models_{\rho} \varphi\}$$

to denote the set of all models of a theory \mathcal{T} . Note that we also have $\mathcal{M}(\mathcal{T}) = \{I \in \mathfrak{S} \mid \forall \varphi \in \mathcal{T} : \forall \rho : I \models_{\rho} \varphi\}$ since φ is a sentence, and if there exists an interpretation I and an environment ρ such that $I \models_{\rho} \varphi$, then for all ρ' , $I \models_{\rho'} \varphi$. This is because the values of variables in ρ/ρ' do not influence the environment of φ .

2.2 The SMT Problem

In this section, we first recall the notion of satisfiability and validity (modulo theory). We also recall the theories of interest in SMT solvers. We then present the abstract DPLL(\mathcal{T}) algorithm for solving the SMT problem. At the end, we recall the theory combination problem in SMT and introduce Nelson-Oppen method.

2.2.1 Satisfiability and Validity

We say a first-order formula φ is *satisfiable* (with respect to the set \mathfrak{S} of interpretations) if and only if there exist an interpretation I and an environment ρ that make the formula *true*:

$$\text{satisfiable}(\varphi) \triangleq \exists I \in \mathfrak{S} : \exists \rho : I \models_{\rho} \varphi.$$

And the *unsatisfiable* is:

$$\text{unsatisfiable}(\varphi) \triangleq \forall I \in \mathfrak{S} : \forall \rho : I \models_{\rho} \neg\varphi.$$

We say a first-order formula φ is *valid* when all interpretations and environments make it *true*:

$$\text{valid}(\varphi) \triangleq \forall I \in \mathfrak{S} : \forall \rho : I \models_{\rho} \varphi.$$

And the *invalid* is:

$$\text{invalid}(\varphi) \triangleq \exists I \in \mathfrak{S} : \exists \rho : I \models_{\rho} \neg\varphi.$$

Hence, we have φ is satisfiable if and only if $\neg\varphi$ is invalid, and φ is valid if and only if $\neg\varphi$ is unsatisfiable.

If we only consider a subset $\mathcal{I} \in \wp(\mathfrak{S})$ of interpretations such that:

$$\text{satisfiable}_{\mathcal{I}}(\varphi) \triangleq \exists I \in \mathcal{I} : \exists \rho : I \models_{\rho} \varphi$$

we call it *satisfiability modulo interpretations* \mathcal{I} . We also have *validity modulo interpretations* \mathcal{I} such as:

$$\text{valid}_{\mathcal{I}}(\varphi) \triangleq \forall I \in \mathcal{I} : \forall \rho : I \models_{\rho} \varphi.$$

Moreover, when there exists a theory \mathcal{T} such that $\mathcal{I} = \mathcal{M}(\mathcal{T})$, we have the notion of *satisfiability and validity modulo theory* \mathcal{T} , where we only consider interpretations $I \in \mathcal{M}(\mathcal{T})$ that are models of the theory. Hence, we have:

$$\text{satisfiable}_{\mathcal{T}}(\varphi) \triangleq \text{satisfiable}_{\mathcal{M}(\mathcal{T})}(\varphi) = \exists I \in \mathcal{M}(\mathcal{T}) : \exists \rho : I \models_{\rho} \varphi$$

$$\text{valid}_{\mathcal{T}}(\varphi) \triangleq \text{valid}_{\mathcal{M}(\mathcal{T})}(\varphi) = \forall I \in \mathcal{M}(\mathcal{T}) : \forall \rho : I \models_{\rho} \varphi$$

which are also called \mathcal{T} -satisfiable and \mathcal{T} -valid.

2.2.2 Theories of interest

One prominent difference between SMT solvers and other first-order logical solvers is that SMT solvers only concentrate on some specialized theories of particular interest instead of focusing on more general methods. Those specialized theories of interest should be expressive enough to model interesting problems and have efficient decision procedures (a terminating algorithm that gives either a yes or a no answer to a formal problem). Below we recall several popular theories used in most state-of-the-art general-purpose SMT solvers:

Equality. Usually, a theory imposes some restrictions on how function or predicate symbols are interpreted. However, the most general case is a theory which imposes no such restrictions, in other words, a theory that includes all possible models for a given signature. Given any signature, we denote the theory that includes all possible models of that theory as \mathcal{T}_ϵ . It is also sometimes called the *empty theory* because its finite axiomatization is just \emptyset . Because no constraints are imposed on the way the symbols in the signature may be interpreted, it is also sometimes called the *theory of equality with uninterpreted functions* (EUF). The satisfiability problem for conjunctions of formulas modulo T_ϵ is decidable in polynomial time using a procedure known as *congruence closure* [BTV03, NO05].

Linear integer arithmetic. Let $\Sigma_{\mathcal{Z}} = \{0, 1, +, -, \leq\}$ be the signature and the theory $\mathcal{T}_{\mathcal{Z}}$ consist of the model that interprets these symbols in the usual way over the integers. This theory is also known as *Presburger arithmetic*. The general satisfiability problem for conjunctions of formulas modulo $\mathcal{T}_{\mathcal{Z}}$ is decidable, but its complexity is triply-exponential.

On the other hand, the satisfiability problem for conjunctions of quantifier-free formulas modulo $\mathcal{T}_{\mathcal{Z}}$ is NP-complete [Pap81].

Linear real arithmetic. Let $\Sigma_{\mathcal{R}} = \{0, 1, +, -, \leq\}$ be the signature and the theory $\mathcal{T}_{\mathcal{R}}$ consist of the model that interprets these symbols in the usual way over the reals. The general satisfiability problem for conjunctions of formulas modulo $\mathcal{T}_{\mathcal{R}}$ is decidable, but its complexity is doubly-exponential. On the other hand, the satisfiability problem for conjunctions of quantifier-free formulas modulo $\mathcal{T}_{\mathcal{R}}$ is polynomial [Kar84], though exponential methods (Simplex) tend to perform best in practice [DdM06].

Arrays. Let $\Sigma_{\mathcal{A}} = \{read, write\}$ be the signature. Given an infinite array a , the term $read(a, i)$ denotes the value at index i of a and the term $write(a, i, v)$ denotes an array that is identical to a except that the value at index i is v . Let $\Lambda_{\mathcal{A}}$ be the following axioms:

$$\forall a : \forall i : \forall v : read(write(a, i, v), i) = v$$

$$\forall a : \forall i : \forall j : \forall v : i \neq j \rightarrow read(write(a, i, v), j) = read(a, j)$$

$$\forall a : \forall b : (\forall i : read(a, i) = read(b, i)) \rightarrow a = b$$

Then the theory $\mathcal{T}_{\mathcal{A}}$ of infinite arrays is the set of all models of these axioms. The general satisfiability problem for conjunctions of formulas modulo $\mathcal{T}_{\mathcal{A}}$ is undecidable. On the other hand, the satisfiability problem for conjunctions of quantifier-free formulas modulo $\mathcal{T}_{\mathcal{A}}$ is NP-complete. In practice, several algorithms have been developed which work well [SJ80, Opp80, SBDL01, BMS06]. The theory of arrays are often used to model memories and to reason about memory reads and writes.

Other useful theories include *fixed-width bitvectors* which can easily encode common CPU arithmetic instructions and *inductive data type* which can be used to model a variety

of things, e.g., enumerations, records, tuples, program data types, and type systems. Moreover, combining theories may provide more expressivity such that the combination of theory of arrays and theory of fixed-width bitvectors gives a natural logic for encoding assembly instructions.

2.2.3 Abstract DPLL(\mathcal{T})

The most common method used for solving SMT problems is DPLL(\mathcal{T}) [GHN⁺04, NOT06, Seb07, BSST09], in which efficient SAT solvers are integrated with decision procedures for first-order theories. In this section, we present abstract DPLL(\mathcal{T}) which describes DPLL(\mathcal{T}) procedure abstractly as a transition system [NOT06].

In abstract DPLL(\mathcal{T}), states are of form *Fail* or $\mu \parallel \varphi$ where φ is a CNF formula and μ is a sequence of literals, each marked as a *decision* or a non-decision literal, which represents a partial assignment of truth values to the atomic formulas (atoms) of φ . We write l^d to denote it as the decision literal. We write $\mu \models_p \varphi$ to mean that if μ is propositionally satisfiable, so does φ . The initial state is $\emptyset \parallel \varphi$. The transition relation is specified by a set of *transition rules* such that:

Definition 2.2.1 (Transition rules of $DPLL(\mathcal{T})$).

$$\textbf{Propagate: } \mu \parallel \varphi \wedge (c \vee l) \implies \mu, l \parallel \varphi \wedge (c \vee l) \text{ if } \begin{cases} \mu \models_p \neg c \\ l \text{ is undefined in } \mu \end{cases}$$

$$\textbf{Decide: } \mu \parallel \varphi \implies \mu, l^d \parallel \varphi \text{ if } \begin{cases} l \text{ or } \neg l \text{ occurs in } \varphi \\ l \text{ is undefined in } \mu \end{cases}$$

$$\textbf{Fail: } \mu \parallel \varphi \wedge c \implies \textit{Fail} \text{ if } \begin{cases} \mu \models_p \neg c \\ \mu \text{ contains no decision literals} \end{cases}$$

$$\textbf{Restart: } \mu \parallel \varphi \implies \emptyset \parallel \varphi$$

$$\textbf{\mathcal{T}-Propagate: } \mu \parallel \varphi \implies \mu, l \parallel \varphi \text{ if } \begin{cases} \mu \models_{\mathcal{T}} l \\ l \text{ or } \neg l \text{ occurs in } \varphi \\ l \text{ is undefined in } \mu \end{cases}$$

$$\textbf{\mathcal{T}-Learn: } \mu \parallel \varphi \implies \mu \parallel \varphi \wedge c \text{ if } \begin{cases} \text{each atoms of } c \text{ occurs in } \mu \parallel \varphi \\ \varphi \models_{\mathcal{T}} c \end{cases}$$

$$\textbf{\mathcal{T}-Forget: } \mu \parallel \varphi \wedge c \implies \mu \parallel \varphi \text{ if } \varphi \models_{\mathcal{T}} c$$

\mathcal{T}-Backjump:

$$\mu, l^d, \mu' \parallel \varphi \wedge c \implies \mu, l' \parallel \varphi \wedge c \text{ if } \begin{cases} \mu, l^d, \mu' \models_p \neg c, \text{ and there is} \\ \text{some clause } c' \vee l' \text{ such that:} \\ \varphi \wedge c \models_{\mathcal{T}} c' \vee l' \text{ and } \mu \models_p \neg c' \\ l' \text{ is undefined in } \mu, \text{ and} \\ l' \text{ or } \neg l' \text{ occurs in } \mu, l^d, \mu' \parallel \varphi \end{cases}$$

□

The clause c in the **\mathcal{T} -Backjump** rule is called *conflicting* clause and $c' \vee l'$ in the same rule is called *backjump* clause. The sequence $S_0 \implies S_1 \implies S_2 \implies \dots$ following the transition rules of $\text{DPLL}(\mathcal{T})$ in definition 2.2.1 is called *derivation*. Moreover, [BDH⁺13] shows that $\text{DPLL}(\mathcal{T})$ is an abstract interpretation of the semantics of logical formulas.

2.2.4 Combining Theories

In SMT, it is common to face with formulas involving several theories in practice, e.g., the combination of theory of integer and theory of arrays. Hence, a natural question is given two theories \mathcal{T}_1 and \mathcal{T}_2 over signatures Σ_1 and Σ_2 , whether it is possible to combine their theory solvers into a single theory solver that decide satisfiability modulo the combination $\mathcal{T}_1 \oplus \mathcal{T}_2$ over the signature $\Sigma_1 \cup \Sigma_2$.

Unfortunately, there does not exist a general complete method for combining any theories due to undecidability. By imposing some restriction on the component theories and their combination, Nelson and Oppen proposed a successful method for combining theory solvers [NO79]. It is fair to say that most work in theory combination in SMT is based on extensions and refinements of Nelson and Oppens work. In this section, we present a declarative non-deterministic combination framework, first presented in [Opp80].

Given two theories \mathcal{T}_1 and \mathcal{T}_2 over signatures Σ_1 and Σ_2 , the Nelson-Oppen method is applicable when:

- the signatures $\Sigma_i, i = 1, 2$ are disjoint, i.e., $\Sigma_1 \cap \Sigma_2 = \emptyset$,
- each theory $\mathcal{T}_i, i = 1, 2$ is *stably infinite* (a theory is stably infinite if every \mathcal{T} -satisfiable formula is satisfiable in infinite models),
- the formulas to be tested for satisfiability are quantifier-free.

The Nelson-Oppen method often begins with a purification procedure which turns the

formula φ into an equisatisfiable pure form $\varphi = \varphi_1 \wedge \varphi_2$ where each φ_i is \mathcal{T}_i -satisfiable. A term t is called *i-term* if its top function symbol is in $\Sigma_i, i = 1, 2$. A literal l is called *i-literal* if its predicate symbol is in $\Sigma_i, i = 1, 2$ or if it is of form $(\neg)s = t$ and both s and t are *i-term*. A subterm of an *i-term* l is an *alien* subterm of l if it is a *j-term*, with $j \neq i$, and all its superterms in l are *i-terms*. An *i-term* or *i-literal* is called *pure* if it only contains symbols from $\Sigma_i, i = 1, 2$. The purification procedure consists of the following steps:

Purification procedure.

1. **Initial state.** Let φ be a conjunction of literals over $\Sigma_1 \cup \Sigma_2$.
2. **Abstract alien subterms.** Replace each alien subterm t of a literal φ with a fresh variable x and add (conjunctively) the equation $x = t$ to φ . Repeat until no more subterms in φ .
3. **Separate.** let $\varphi_i, i = 1, 2$ be the conjunctions of all the *i-literals* in (the new) φ .

Then the combination procedure of the Nelson-Oppen method consists of the following steps:

Combination procedure.

1. **Initial state.** Let φ be a conjunction of literals over $\Sigma_1 \cup \Sigma_2$.
2. **Purification.** Preserving satisfiability transform φ into $\varphi_1 \wedge \varphi_2$ where $\varphi_i, i = 1, 2$ are the conjunctions of the *i-literals*.
3. **Guess a partition.** Let \varkappa be the set of all variables occurs in both φ_1 and φ_2 , $S \triangleq \{(x_1, x_2) \mid x_1, x_2 \in \varkappa\}$, guess a partition $E \cup I$ on S where $x_1 = x_2$ when

$(x_1, x_2) \in E$ and $x_1 \neq x_2$ when $(x_1, x_2) \in I$. Let

$$\psi \triangleq \bigwedge_{(x_1, x_2) \in E} (x_1 = x_2) \wedge \bigwedge_{(x_1, x_2) \in I} (x_1 \neq x_2)$$

4. **Check pure formulas.** Check whether $\varphi_i \wedge \psi$ is \mathcal{T}_i -satisfiable for $i = 1, 2$.
5. **Return.** Return satisfiable when both return yes; return unsatisfiable otherwise.

In practice, the purification is unnecessary if each theory solver accepts literals containing alien subterms, and treats the latter as if they were free constants. A Nelson-Oppen procedure without the purification step can be found in [BDS02]. By imposing several restrictions on the component theories, it is also possible to deduct the equalities to be shared instead of guessing, see [NO03, NO05, LM05]. Moreover, Cousot, Cousot and Mauborgne [CCM11] shows that the Nelson-Oppen procedure is a reduced product.

2.3 SMT-LIB

SMT-LIB, short for Satisfiability modulo theories library, is an international initiative aimed at facilitating research and development in SMT. It provides a standard rigorous description of background theories used in SMT systems and a common input and output language for SMT solvers. The latest version of SMT-LIB is version 2.0. In this section, we give a brief introduction of theories, logics and commands of its language referred in SMT-LIB v2 [BST10].

2.3.1 Theories

SMT-LIB includes the following theories which the SMT-LIB logics refer to one or more of them:

Core. The Core theory defines the basic Boolean operators which include two constant symbols `true` and `false` representing Boolean values *true* and *false*, several function symbols `not`, `and`, `or`, `xor` and `=>` (implies) representing standard Boolean operators, and three predicate symbols `=`, `distinct` and `ite`. The predicate `=` has two arguments which returns `true` when its two arguments are identical while the predicate `distinct` also has two arguments but returns `true` when its two arguments are not identical. The predicate `ite` has three arguments where its first argument is a predicate. It returns its second argument when its first argument evaluates to `true` or returns its third argument when its first argument evaluates to `false`.

Ints. The Ints theory defines the theory of linear integer arithmetic. Its signature includes all numerals and all terms of the form $(- n)$ where n is a numeral other than 0 as its constant symbols. It includes several function symbols such as `+`, `-`, `*`, `div`, `mod`, `abs` which representing standard integer arithmetic operators and `<`, `<=`, `>`, `>=` which representing standard integer comparison operators.

Reals. The Reals theory defines the theory of linear real arithmetic. Its constant symbols include an abstract value for each irrational algebraic number, all numerals, all terms of the form $(- n)$ where n is a numeral other than 0 and all terms of the form $(/ m n)$ or $(/ (- m) n)$ where m is a numeral other than 0, n is a numeral other than 0 and 1, m and n have no common factors besides 1. Its function symbols include `+`, `-`, `*`, `div`, `mod`, `abs` which representing standard real arithmetic operators and `<`, `<=`, `>`, `>=` which representing standard real comparison operators.

Reals_Ints. The Reals_Ints theory defines the combination of theory of linear integer arithmetic and theory of linear real arithmetic. It includes all constants symbols and function symbols defined in the ints theory and reals theory. It also defines three other

function symbols `to_real`, `to_int` and `is_int`. The function `to_real` transforms an integer into the real number while the function `to_int` maps a real number to its integer part. The function `is_int` tests whether a real number is an integer. Note that the mix of integers and real numbers in one literal is not allowed. The user must explicit convert all integers to real numbers using `to_real` or convert all real numbers to integers using `to_int`.

ArrayEx. The `ArrayEx` theory defines the theory of functional arrays with extensionality. It defines two function symbols `select` and `store` whose meanings are the same as in 2.2.2. The `ArrayEx` theory should also satisfy all axioms defined in 2.2.2.

FixedSizeBitVectors. The `FixedSizeBitVectors` theory defines the theory of bit vectors with arbitrary size. The bit vectors are defined as binaries (`#bX BitVec m`) where `m` is the number of digits in `X` and hexdeximals (`#xX BitVec m`) where `m` is 4 times the number of digits in `X`. The `FixedSizeBitVectors` theory defines two function symbols `concat` and `extract` which representing the operations of concatenation and extraction of bit vectors. It also defines usual bit-wise operators including `bvnot`, `bvand`, `bvor`, arithmetic operators including `bvneg`, `bvadd`, `bvmul`, `bvudiv`, `bvurem` and shift operators `bvshl`, `bvshr`.

2.3.2 Logics

SMT-LIB defines a lot of logics, from the sub-logics to the combination of two or more of them. Dividing logics into sub-logics helps identifying fragments of the main logic where it may possible to apply specialized and more efficient satisfiability techniques. In this section, we will recall several (sub)-logics we are interested in.

AUFLIA: Closed formulas over the theory of linear integer arithmetic and arrays extended with free function symbols but restricted to arrays with integer indices and values.

AUFLIRA: Closed linear formulas with free function symbols over one and two dimensional arrays of integer index and real value.

AUFNIRA: Closed formulas with free function and predicate symbols over a theory of arrays of integer index and real value.

LRA: Closed linear formulas in linear real arithmetic.

QF_AUFLIA: Closed quantifier-free linear formulas over the theory of integer arrays extended with free function symbols.

QF_AX: Closed quantifier-free formulas over the theory of arrays with extensionality.

QF_LIA: Unquantified linear integer arithmetic. In essence, Boolean combinations of inequation between linear polynomials over integer variables.

QF_LRA: Unquantified linear real arithmetic. In essence, Boolean combinations of inequation between linear polynomials over real variables.

QF_NIA: Quantifier-free integer arithmetic.

QF_NRA: Quantifier-free real arithmetic.

QF_UF: Unquantified formulas built over a signature of uninterpreted (i.e., free) function symbols.

QF_UFLIA: Unquantified linear integer arithmetic with uninterpreted function symbols.

QF_UFLRA: Unquantified linear real arithmetic with uninterpreted function symbols.

UFLRA: Linear real arithmetic with uninterpreted function symbols.

2.3.3 SMT2 Commands

SMT-LIB defines a large set of commands which help users to write standard SMT scripts that can be accepted by every SMT solvers supporting it. In this section, we recall a key subset of the SMT2 commands. More details about those commands can be found in [BST10].

(set-logic L): This command tells the solvers what logic will be used. The argument L can be the name of a logic defined in the SMT-LIB or of some other solver-specific logic. If this is left unspecified, the solvers can assume an ALL SUPPORTED logic that enables the widest range of supported formulas or just simply return **error**. This command must precede all of the other commands in this list.

(declare-fun $f \sigma^* \sigma$): This command declares an uninterpreted function symbol f that operates over a list of arguments of types σ^* and the returns a value of type σ . Variables are declared as uninterpreted constants by leaving the domain list empty, e.g. **(declare-fun x () Int)**.

(assert t): This command adds term t to the assertion set on the top of the assertion-set stack if t is a closed formula (i.e., a well sorted closed term of sort **Int**). Otherwise, it returns an error.

(check-sat): This command instructs the solver to check whether the conjunction of all the formulas in the set of all assertions is satisfiable in the logic specified with the `set-logic` command. It will return either `sat` which indicates that the solver has found a model, `unsat` which indicates that the solver has established there is no model, and `unknown` which indicates that the search was inconclusive - because of time limits, solver incompleteness, and so on.

(get-model): After the `(check-sat)` call with the answer `sat`, this command asks the solver to return a satisfying interpretation.

(get-value t_1, \dots, t_n): Similar to the command `(get-model)`, this command asks the solver to return a satisfying interpretation of the given terms t_1, \dots, t_n instead of the whole model.

The following example gives a very simple SMT script that asks if there is a solution to the pair of equations $x + 2 * y = 20$ and $x - y = 2$:

Example 2.3.1. Simple SMT-LIB example.

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= (+ x (* 2 y)) 20))
(assert (= (- x y) 2))
(check-sat)
(get-model)
```

The solver will return `sat`, indicating that the formula is satisfiable. The solver will also return a model `((x 8) (y 6))`, indicating $x = 8$ and $y = 6$ is a satisfying solution.

Part II

SMT-based Abstract Domains

Chapter 3

Affine Equalities

In this chapter, we introduce a logical abstract domain whose elements are logical formulas involving finite conjunctions of affine equalities. An affine equality is of form $a_1x_1 + a_2x_2 + \dots + a_nx_n = b$. We assume $a_i, b, x_i, 1 \leq i \leq n$ are all integers.

3.1 Normal Form of Affine Equalities

Given a set of affine equalities of n variables:

$$\left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2, \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m \end{array} \right.$$

Its normal form will be:

$$\left\{ \begin{array}{l} c_{11}x_1 + c_{12}x_2 + \cdots c_{1n}x_n = d_1, \\ c_{21}x_1 + c_{22}x_2 + \cdots c_{2n}x_n = d_2, \\ \vdots \\ c_{l1}x_1 + c_{l2}x_2 + \cdots c_{ln}x_n = d_l \end{array} \right.$$

where $l \leq m$ and c_{ij} satisfies:

- (1) $\forall i \in [1, l] : \exists j \in [1, n] : c_{ij} \neq 0.$
- (2) For each $i_0 \in [1, l]$, if $c_{i_0 j_0} \neq 0$ and $\forall j < j_0, c_{i_0 j} = 0$, then $\forall i > i_0 : \forall j \leq j_0 : c_{ij} = 0.$

This normal form is also called normalized reduced row-echelon form [Kar76] and the proof of its existence can be found in [Mun64]. It follows that the normal form of a set of affine equalities of n variables has at most n affine equalities.

Normally, row operations, such as multiply a row by a non-zero scalar, or add or subtract a row to another row, or permute two rows, may be used to transform a set of affine equalities to its normal form. In this section, we are going to discuss how to use SMT solvers for the transformations. A set of affine equalities can be easily written into a first-order logical formula such as:

$$\varphi = \bigwedge_{i \in [1, m]} a_{i1}x_1 + a_{i2}x_2 + \cdots a_{in}x_n = b_i$$

3.1.1 Generation from Points P

Given a set of n -dimensional points P where $|P| = m, m \geq 1$, we can generate the normal form of a set of affine equalities, which satisfies each point in P , directly by using SMT solvers. Note that when $m = 1$, the normal form of the set of affine equalities satisfies

the point p can be written directly as:

$$\varphi = \bigwedge_{i \in [1, n]} x_i = p_i \quad (3.1)$$

When $m > 1$, let's consider the following template equality:

$$c_1 x_1 + c_2 x_2 + \dots + c_n x_n = d \quad (3.2)$$

When we replace $x_i, 1 \leq i \leq n$ by the points in P , we will get a set of equalities:

$$\left\{ \begin{array}{l} p_{11}c_1 + p_{12}c_2 + \dots + p_{1n}c_n = d, \\ p_{21}c_1 + p_{22}c_2 + \dots + p_{2n}c_n = d, \\ \vdots \\ p_{m1}c_1 + p_{m2}c_2 + \dots + p_{mn}c_n = d \end{array} \right. \quad (3.3)$$

where c_1, \dots, c_n, d are unknowns. By rewriting the above equalities into a first-order logical formula, we can then ask an SMT solver if this formula is satisfiable under the theory of integer (or real arithmetic). If satisfiable, we can ask the SMT solvers to return a possible non-zero model for c_1, \dots, c_n, d . This gives us one possible equality that satisfies P . Because SMT solvers only generate arbitrary models, so we must specify what kind of c_1, \dots, c_n, d we need to get the normal form of affine equalities that satisfy P .

From the definition of normal form, we can generate the following set of possible c_1, \dots, c_n, d :

$$\left\{ \begin{array}{l} c_1 \neq 0, c_2, \dots, c_n, d \text{ are integers,} \\ c_1 = 0, c_2 \neq 0, c_3, \dots, c_n, d \text{ are integers,} \\ \vdots \\ c_1 = 0, c_2 = 0, \dots, c_{n-1} = 0, c_n \neq 0, d \text{ is an integer.} \end{array} \right. \quad (3.4)$$

Then all satisfiable c_1, \dots, c_n, d forms the normal form of affine equalities that contains all points in P . If there is no satisfiable c_1, \dots, c_n, d , then return \top . Moreover, to make the models generated by SMT solvers easily comparing to each other, we can also use the following set of possible c_1, \dots, c_n, d :

$$\left\{ \begin{array}{l} c_1 = 1, c_2, \dots, c_n, d \text{ are rational numbers,} \\ c_1 = 0, c_2 = 1, c_3, \dots, c_n, d \text{ are rational numbers,} \\ \vdots \\ c_1 = 0, c_2 = 0, \dots, c_{n-1} = 0, c_n = 1, d \text{ is a rational number.} \end{array} \right. \quad (3.5)$$

Example 3.1.1. Let $P = \{(1, 2, 3, 4, 5), (2, 3, 4, 5, 6)\}$, we then have

$$\left\{ \begin{array}{l} c_1 + 2c_2 + 3c_3 + 4c_4 + 5c_5 = d \\ 2c_1 + 3c_2 + 4c_3 + 5c_4 + 6c_5 = d \end{array} \right.$$

We first translate it into the following SMT script:

```
(set-logic QF_LRA)
(declare-fun c1 () Real)
(declare-fun c2 () Real)
(declare-fun c3 () Real)
(declare-fun c4 () Real)
(declare-fun c5 () Real)
(declare-fun d () Real)
(assert (= (+ c1 (+ (* 2 c2) (+ (* 3 c3) (+ (* 4 c4)
(* 5 c5)))))) d))
(assert (= (+ (* 2 c1) (+ (* 3 c2) (+ (* 4 c3) (+ (* 5 c4)
(* 6 c5)))))) d))
```

Note that the real numbers in the SMT solvers are actually represented as rational numbers. Then we translate “ $c_1 = 1, c_2, \dots, c_n, d$ are rational numbers” by adding the following SMT script:

```
(assert (= c1 1))

(check-sat)

(get-model)
```

By calling Z3 [dMB08], it will return “sat” and produce the model which is “ $c_1 = 1, c_2 = -3/2, c_3 = 0, c_4 = 1/2, c_5 = 0, d = 0$ ”. Continuing to check other possible c_1, \dots, c_n, d , we then get

- $c_1 = 0, c_2 = 1, c_3 = -3/2, c_4 = 0, c_5 = 1/2, d = 0$
- $c_1 = 0, c_2 = 0, c_3 = 1, c_4 = 0, c_5 = -1, d = -2$
- $c_1 = 0, c_2 = 0, c_3 = 0, c_4 = 1, c_5 = -1, d = -1$
- “unsat” for $c_1 = 0, c_2 = 0, c_3 = 0, c_4 = 0, c_5 = 1$

After several adjustment, such as multiple by dividend, we have the affine equalities in normal form that satisfy P :

$$\left\{ \begin{array}{l} 2x_1 - 3x_2 + x_4 = 0 \\ 2x_2 - 3x_3 + x_5 = 0 \\ x_3 - x_5 = -2 \\ x_4 - x_5 = -1 \end{array} \right.$$

□

Note that different SMT solver may generate different results for c_1, \dots, c_n, d . For

example, the normal form generated by CVC4 [BCD⁺11] will be:

$$\begin{cases} x_1 - 2x_2 + x_3 = 0 \\ x_2 - 2x_3 + x_4 = 0 \\ x_3 - 2x_4 + x_5 = 0 \\ x_4 - x_5 = -1 \end{cases}$$

Hence, all the examples in the following chapters, unless clearly stated, will use Z3 as our SMT solver.

3.1.2 Normalization

Given any set of affine equalities F of n variables, we can easily transform it into the normal form using an SMT solvers. Let $F_i^N, 1 \leq i \leq n$ be the normal form of affine equalities generated from i independent points. We say n points are independent if and only if for any subset of these points, the normal form of affine equalities generated from them will not satisfy any other points. Then the normalization can be done as follow:

0. Check the satisfiability of F using an SMT solver. If “unsat”, return \perp . Otherwise, ask the SMT solver to generate a model for F , let it be p_1 .
1. Generating F_1^N from $\{p_1\}$. Check the satisfiability of $F \wedge \neg F_1^N$ using the SMT solver. If “unsat”, then F_1^N is the normal form of F . Otherwise, ask the SMT solver to generate a model for $F \wedge \neg F_1^N$, let it be p_2 .
2. Generating F_2^N from $\{p_1, p_2\}$. Check the satisfiability of $F \wedge \neg F_2^N$ using the SMT solver. If “unsat”, then F_2^N is the normal form of F . Otherwise, ask the SMT solver to generate a model for $F \wedge \neg F_2^N$, let it be p_3 .
- \vdots

- i. Generating F_i^N from $\{p_1, \dots, p_i\}$. Check the satisfiability of $F \wedge \neg F_i^N$ using the SMT solver. If “unsat”, then F_i^N is the normal form of F . Otherwise, ask the SMT solver to generate a model for $F \wedge \neg F_i^N$, let it be p_{i+1} .
- ⋮

Note that this procedure will have at most $n+1$ steps, hence it will guarantee to terminate.

We call the set of points $\{p_1, \dots, p_i, \dots\}$ the *normalization points* of F .

Example 3.1.2. Given the following set of affine equalities:

$$\begin{cases} x_1 + x_2 + x_3 = 0 \\ x_1 + 2x_2 + 3x_3 = 0 \end{cases}$$

We first translate it into the SMT script:

```
(set-logic QF_LIA)
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(assert (= (+ x1 (+ x2 x3)) 0))
(assert (= (+ x1 (+ (* 2 x2) (* 3 x3))) 0))
(check-sat)
(get-model)
```

By calling Z3, it will return “sat” and generate the model “x1 = 1, x2 = -2, x3 = 1”. We then check with the following SMT script:

```
(assert (not (and (= x1 1) (and (= x2 -2) (= x3 1)))))
```

Z3 will still return “sat” and generate a new model “x1 = -1, x2 = 2, x3 = -1”. We generate the normal form of affine equalities from $\{(x_1 = 1, x_2 = -2, x_3 = 1), (x_1 = -1,$

$x_2 = 2, x_3 = -1$). It will be:

$$F = \begin{cases} x_1 - x_3 = 0 \\ x_2 + 2x_3 = 0 \end{cases}$$

We then check with the following SMT script:

```
(assert (not (= (- x1 x3) 0)))
(assert (not (= (+ x2 (* 2 x3)) 0)))
```

Z3 will return “unsat” which means we get the normal form F . □

3.2 SMT-Based Affine Equality Abstract Domain

In this section, we introduce the SMT-based affine equality abstract domain which is able to represent and manipulate the invariants in the form of the finite conjunctions of affine equalities. SMT solvers will be used for the computation of transformations and other logical operations in the domain.

3.2.1 Representation

Let $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$ be the finite set of program variables. We use the finite conjunctions of affine equalities

$$\varphi = \bigwedge_{i \in [1, m]} a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n = b_i$$

to represent the relational properties of \mathcal{X} . We prefer every finite conjunction of affine equalities to be in normal form, but it is not necessary. The set of all possible values of \mathcal{X} by a finite conjunction of affine equalities is given by the following concretization

function:

Definition 3.2.1. The concretization of a finite conjunction of affine equalities is

$$\gamma^e(\varphi) \triangleq \{(v_1, v_2, \dots, v_n) \mid (v_1, v_2, \dots, v_n) \models \varphi\}$$

□

We use *false* to represent the abstract infimum \perp^e and *true* to represent the abstract supremum \top^e .

3.2.2 Binary Operations

We now introduce several useful abstract binary operators for the SMT-based affine equality abstract domain which are necessary to define an abstract domain.

Inclusion Testing

We can easily compare two finite conjunctions of affine equalities using the SMT solver.

Theorem 3.2.1 (Inclusion test for finite conjunctions of affine equalities). *Given two finite conjunctions of affine equalities φ_1 and φ_2 , we have*

$$\varphi_1 \sqsubseteq^e \varphi_2 \triangleq \varphi_1 \models \varphi_2 = \forall \mathcal{X} : \varphi_1 \Rightarrow \varphi_2.$$

□

In practice, we prefer less universal quantifications in the formulas when checking them with SMT solvers. Besides, we know that $\forall x : \varphi \equiv \neg \exists x : \neg \varphi$. Hence, we have $\forall \mathcal{X} : \varphi_1 \Rightarrow \varphi_2 = \neg \exists \mathcal{X} : \neg(\varphi_1 \Rightarrow \varphi_2) = \neg \exists \mathcal{X} : \neg(\neg \varphi_1 \vee \varphi_2) = \neg \exists \mathcal{X} : \varphi_1 \wedge \neg \varphi_2$. We

then use SMT solvers to check the satisfiability of $\varphi_1 \wedge \neg\varphi_2$. If “unsat”, we know that $\varphi_1 \sqsubseteq^e \varphi_2$, otherwise, we have $\varphi_1 \not\sqsubseteq^e \varphi_2$.

Equality Testing

Equality testing is similar to the inclusion testing. We have the following theorem:

Theorem 3.2.2 (Equality testing of finite conjunctions of affine equalities). *Given two finite conjunctions for affine equalities φ_1 and φ_2 , we have*

$$\varphi_1 = \varphi_2 \triangleq \forall \mathcal{X} : \varphi_1 \Leftrightarrow \varphi_2$$

□

Hence, we can either check $\forall \mathcal{X} : \varphi_1 \Leftrightarrow \varphi_2$ directly. In practice, we may check $\varphi_1 \sqsubseteq^e \varphi_2$ and $\varphi_2 \sqsubseteq^e \varphi_1$ instead since $\varphi_1 = \varphi_2 \triangleq \varphi_1 \sqsubseteq^e \varphi_2 \wedge \varphi_2 \sqsubseteq^e \varphi_1$. In the later case, if both return true, then we have $\varphi_1 = \varphi_2$, otherwise, we have $\varphi_1 \neq \varphi_2$.

Meet

The meet (or intersection) of two finite conjunctions of affine equalities is actually a conjunction of all affine equalities.

Theorem 3.2.3 (Meet of finite conjunctions of affine equalities). *Given two finite conjunctions of affine equalities φ_1 and φ_2 , we have*

$$\varphi_1 \sqcap^e \varphi_2 \triangleq \varphi_1 \wedge \varphi_2$$

□

Hence, the only thing left is to normalize $\varphi_1 \wedge \varphi_2$. Let it be φ , we then have $\varphi_1 \sqcap^e \varphi_2 = \varphi$.

Join

The join (or union) of two finite conjunctions of affine equalities is actually a disjunction of them.

Theorem 3.2.4 (Join of finite conjunctions of affine equalities). *Given two finite conjunctions of affine equalities φ_1 and φ_2 , we have*

$$\varphi_1 \sqcup^e \varphi_2 \triangleq \varphi_1 \vee \varphi_2$$

□

Unlike the meet, there may not exist a finite conjunction of affine equalities φ that makes $\forall \mathcal{X} : \varphi \Leftrightarrow \varphi_1 \vee \varphi_2$. Instead, we calculate a finite conjunction of affine equalities φ' which satisfies $\forall \mathcal{X} : \varphi_1 \vee \varphi_2 \Rightarrow \varphi'$ and $\forall \mathcal{X} : \forall \varphi'' : \varphi_1 \vee \varphi_2 \Rightarrow \varphi' \wedge \varphi' \Rightarrow \varphi''$. We can calculate such finite conjunction of affine equalities by the normalization procedure letting $F = \varphi_1 \vee \varphi_2$. If we know the normalization points of both φ_1 and φ_2 , let them be P_1 and P_2 , we can also generate φ from $P = P_1 \cup P_2$.

Example 3.2.1. Let $\varphi_1 = x_1 = 1 \wedge x_2 = -2 \wedge x_3 = 1$ and $\varphi_2 = x_1 = -1 \wedge x_2 = 2 \wedge x_3 = -1$, then $\varphi_1 \sqcup^e \varphi_2 = x_1 - x_3 = 0 \wedge x_2 + 2x_3 = 0$. □

3.2.3 Transfer Functions

We now present the transfer functions for the SMT-based affine equality abstract domain.

Assignment Transfer Function

There are two major types of linear assignments, invertible and non-invertible. We first consider the invertible assignment.

The *invertible assignment* has the form

$$x_i = a_1x_1 + \dots + a_ix_i + \dots + a_nx_n + b, \quad a_i \neq 0 \quad (3.6)$$

The fact $a_i \neq 0$ means we can carry over the knowledge of the previous value of x_i to the new value of x_i . Thus we can rewrite it as:

$$x_i = [x_i - (a_1x_1 + \dots + a_{i-1}x_{i-1} + a_{i+1}x_{i+1} + \dots + a_nx_n + b)]/a_i \quad (3.7)$$

Let $\varphi[x \leftarrow e]$ represents the replacement of x by e in φ , then the assignment transfer function for invertible assignment can be defined as:

$$\begin{aligned} & f_i^e \llbracket x_i = a_1x_1 + \dots + a_ix_i + \dots + a_nx_n + b \rrbracket \varphi \\ = & \varphi[x_i \leftarrow [x_i - (a_1x_1 + \dots + a_{i-1}x_{i-1} + a_{i+1}x_{i+1} + \dots + a_nx_n + b)]/a_i] \end{aligned}$$

Example 3.2.2. let $\varphi = x_1 - x_3 = 0 \wedge x_2 + 2x_3 = 0$, given an assignment $x_2 = x_1 - x_2 + x_3$, then $f_i^e \llbracket x_2 = x_1 - x_2 + x_3 \rrbracket = x_1 - x_3 = 0 \wedge x_1 - x_2 + 3x_3 = 0$. \square

The *non-invertible assignment* has the form

$$x_i = a_1x_1 + \dots + a_{i-1}x_{i-1} + a_{i+1}x_{i+1} + \dots + a_nx_n + b \quad (3.8)$$

It appears we have to “loose” some information of old x_i by the assignment to x_i . The simple way to do this is to remove all affine equalities which contain x_i . A better way is:

1. Let l be the last affine equality in φ which contain x_i . Rewrite it into $x_i = (b - \sum a_j x_j)/a_i$.
2. Replace x_i by $(b - \sum a_j x_j)/a_i$ in each affine equality in φ except l .
3. Remove l from φ and call the result φ' .

Then the assignment transfer function for non-invertible assignment can be defined as:

$$\begin{aligned}
& f_n^e[[a_1x_1 + \dots + a_{i-1}x_{i-1} + a_{i+1}x_{i+1} + \dots + a_nx_n + b]] \\
= & \varphi' \wedge x_i = a_1x_1 + \dots + a_{i-1}x_{i-1} + a_{i+1}x_{i+1} + \dots + a_nx_n + b
\end{aligned}$$

Example 3.2.3. let $\varphi = x_1 - x_3 = 0 \wedge x_2 + 2x_3 = 0$, given an assignment $x_2 = x_1 + x_3$, then $f_i^e[[x_2 = x_1 + x_3]] = x_1 - x_3 = 0 \wedge x_2 = x_1 + x_3$. \square

Test Transfer Function

Given a test ϕ , if it has the form $\bigwedge_{i \in [1, m]} a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n = b_i$, then the test transfer function will be:

$$f_t^e[[\phi]]\varphi = \begin{cases} \varphi \sqcap^e \phi & \text{when evaluating in the true branch} \\ \varphi & \text{when evaluating in the false branch} \end{cases}$$

Otherwise, we simply return φ for both true and false branches.

3.2.4 Example

Let's consider the following program:

```

[P0] x = 2; y = 3; z = 5;
[P1] while [P2] (...) {
    [P3] x = x + 1; y = y + 2; z = z + 3;
[P4] }
```

The test does not have the form $\bigwedge_{i \in [1, m]} a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n = b_i$ hence it is not taken into account. Initially, P_0^0 is *true* and $P_1^0, P_2^0, P_3^0, P_4^0$ are *false*. Then each assertion is propagated through the analysis of the program:

$$P_0^1 = \text{true}$$

$$P_1^1 = f_i^e[z = 5](f_i^e[y = 3](f_i^e[x = 2]P_0^1)) = x = 2 \wedge y = 3 \wedge z = 5$$

$$P_2^1 = P_1^1 \sqcup^e P_4^1 = P_1^1 \vee \text{false} = P_1^1$$

$$P_3^1 = P_2^1$$

$$P_4^1 = f_i^e[z = z + 3](f_i^e[y = y + 2](f_i^e[x = x + 1]P_3^1)) = x = 3 \wedge y = 5 \wedge z = 8$$

The assertion P_0 and P_1 will not be changed, hence we continue with P_2 :

$$P_2^2 = P_1^1 \sqcup^e P_4^1 = 3x - z = 1 \wedge 3y - 2z = -1$$

$$P_3^2 = P_2^2$$

$$P_4^2 = f_i^e[z = z + 3](f_i^e[y = y + 2](f_i^e[x = x + 1]P_3^2)) = 3x - z = 1 \wedge 3y - 2z = -1$$

Then $P_1^1 \sqcup^e P_4^2 = 3x - z = 1 \wedge 3y - 2z = -1$ is equal to P_2^2 so that the program analysis has converged.

The final result is:

[P_0 : true] $x = 2$; $y = 3$; $z = 5$;
 [P_1 : $x = 2 \wedge y = 3 \wedge z = 5$] while [P_2 : $3x - z = 1 \wedge 3y - 2z = -1$] (\dots) {
 [P_3 : $3x - z = 1 \wedge 3y - 2z = -1$] $x = x + 1$; $y = y + 2$; $z = z + 3$;
 [P_4 : $3x - z = 1 \wedge 3y - 2z = -1$] }

3.2.5 Implementation

We have implemented our SMT-Based affine equality abstract domain using OCaml. It accepts finite conjunctions of affine equalities. We translate formulas and SMT queries into SMT-LIB v2 scripts hence all SMT solvers which accept SMT-LIB v2 script as their input can be used in our implementation.

We have evaluated our implementation using different SMT solvers, such as cvc4 and z3. We have also compared our implementation with the same operators in the Apron

library. Table 3.1 shows the differences between our implementation using `cvc4`, `z3` and `Apron` when evaluating the program in Sect. 3.2.4. Each column in the table shows the time usage for calculating the assertion (normally involving one abstract operation) in Sect. 3.2.4.

	P_1^1	P_2^1	P_4^1	P_2^2	P_4^2	$P_1^1 \sqcup^e P_4^2$
Z3	0.000457s	0.273667s	0.000229s	0.353616s	0.000256s	0.327332s
CVC4	0.000366s	0.235393s	0.000227s	0.464791s	0.000261s	0.385006s
Apron	0.000106s	0.000134s	0.000075s	0.000184s	0.000089s	0.000137s

Table 3.1: Comparing SMT-based affine equality abstract domain and Equality abstract domain in `Apron` abstract domain library

From Table 3.1, we can see that there are not too much time differences between using `CVC4` and `Z3`. But when comparing to the equality abstract domain in `Apron` abstract domain library, we can find huge time differences from our SMT-based affine equality abstract domain, especially for computing P_2^1 , P_2^2 and $P_1^1 \sqcup^e P_4^2$ which are the computations of the join operators. First, we must admit that our implementation is not optimal. When computing the assignment transfer functions in P_1^1 , P_4^1 and P_4^2 , even we are actually using the same methods as those in `Apron`, our domain is always slower than `Apron`'s. Another reason for such differences is because the join operator in our domain uses the normalization process which needs to call the SMT solvers several times. For example, when computing P_2^2 , we have to call the SMT solvers about 10 times. Hence, if we can arrange these calling into parallel, we can definitely reduce the time of our domain. However, although this may reduce the time of computing the join operators around 10 times lower, there still exists a huge difference between our domain and the abstract domain in `Apron`, e.g., 0.03536s v.s. 0.00018s. Most of these time are used for the SMT solvers to do the solving and generating the models. Hence, we can conclude that the SMT solvers themselves are slower than `Apron`. To improve the performance of our domain, it mainly depends on the improvement of the SMT solvers. With the evolution of

the SMT solvers, we believe our SMT-based affine equality abstract domain will catch up the classical convex abstract domains and will provide a different perspective in designing the new abstract domains.

Chapter 4

Linear Inequalities

In this chapter, we introduce another SMT-based logical abstract domain which involves finite conjunctions of linear inequalities. A linear inequality is of form $a_1x_1 + a_2x_2 + \dots + a_nx_n \leq b$. We assume $a_i, b, x_i, 1 \leq i \leq n$ are all integers.

4.1 Representation

Let $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$ be the finite set of program variables. We use the finite conjunctions of linear inequalities

$$\varphi = \bigwedge_{i \in [1, m]} a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n \leq b_i$$

to represent the relational properties of \mathcal{X} . Strict inequalities will be over-approximated to non-strict ones, e.g., $a_1x_1 + \dots + a_nx_n < b$ will be rewritten to $a_1x_1 + \dots + a_nx_n \leq b$. Linear equalities will be rewritten to conjunctions of two linear inequalities, e.g., $a_1x_1 + \dots + a_nx_n = b$ will be rewritten to $a_1x_1 + \dots + a_nx_n \leq b \wedge (-a_1x_1) + \dots + (-a_nx_n) \leq -b$.

The set of all possible values of \mathcal{X} by a finite conjunction of linear inequalities is given

by the following concretization function:

Definition 4.1.1. The concretization of a finite conjunction of linear inequalities is

$$\gamma^i(\varphi) \triangleq \{(v_1, v_2, \dots, v_n) \mid (v_1, v_2, \dots, v_n) \models \varphi\}$$

□

We use *false* to represent the abstract infimum \perp^i and *true* to represent the abstract supremum \top^i .

4.1.1 Simplification

Given any finite conjunction of linear inequalities, it is often the case that eliminating several linear inequalities will not change its concretization. We call these linear inequalities *irrelevant*. With SMT solvers, we can simplify any finite conjunction of linear inequalities by eliminating irrelevant linear inequalities thanks to the following theorem:

Theorem 4.1.1. Let $\varphi = \ell_1 \wedge \ell_2 \wedge \dots \wedge \ell_m$ where $\ell_i, i \in [1, m]$ is linear inequality, ℓ_j is irrelevant if and only if $\forall \mathcal{X} : \ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m \Rightarrow \ell_j$.

Proof. $\gamma^i(\ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m) = \gamma^i(\varphi)$

$$\iff \{(v_1, v_2, \dots, v_n) \mid (v_1, v_2, \dots, v_n) \models \ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m\} = \{(v_1, v_2, \dots, v_n) \mid (v_1, v_2, \dots, v_n) \models \varphi\}$$

{by definition of $\gamma^i(\varphi) = \{(v_1, v_2, \dots, v_n) \mid (v_1, v_2, \dots, v_n) \models \varphi\}$ }

$$\iff \{(v_1, v_2, \dots, v_n) \mid (v_1, v_2, \dots, v_n) \models \ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m \Leftrightarrow \varphi\}$$

{since $\{x \mid P(x)\} = \{y \mid Q(y)\}$ iff $P(x) \Leftrightarrow Q[y/x]$ }

$$\iff \forall \mathcal{X} : \ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m \Leftrightarrow \varphi \quad \{\text{definition of } \models\}$$

$$\iff \forall \mathcal{X} : \ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m \Rightarrow \ell_j$$

$$\{\phi \Rightarrow \phi \wedge \ell_j \iff \phi \Rightarrow \ell_j \text{ where } \phi = \ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m\}$$

□

Checking $\forall \mathcal{X} : \ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m \Rightarrow \ell_j$ is simple when using SMT solvers.

We first rewrite it as:

$$\begin{aligned} & \forall \mathcal{X} : \ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m \Rightarrow \ell_j \\ = & \forall \mathcal{X} : \neg(\ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m) \vee \ell_j \\ = & \forall \mathcal{X} : \neg(\ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m \wedge \neg \ell_j) \\ = & \neg \exists \mathcal{X} : \ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m \wedge \neg \ell_j \end{aligned}$$

Then we check the satisfiability of $\ell_1 \wedge \dots \wedge \ell_{j-1} \wedge \ell_{j+1} \wedge \dots \wedge \ell_m \wedge \neg \ell_j$. If SMT solvers return “unsat”, which means ℓ_j is irrelevant, we can safely eliminate it from φ . Otherwise, we can’t eliminate ℓ_j . Repeating this procedure on the remaining linear inequalities until no more irrelevant ones, we then obtain a minimal finite conjunction of linear inequalities (with no irrelevant inequalities) corresponding to the same concretization.

4.2 Binary Operations

We introduce several useful binary operators, such as inclusion testing, equality testing, meet and join which are necessary to define an abstract domain.

4.2.1 Inclusion Testing

Comparing two finite conjunctions of linear inequalities using SMT solvers is similar to compare two finite conjunctions of affine equalities. We have the following theorem:

Theorem 4.2.1 (Inclusion test for finite conjunctions of linear inequalities). *Given two finite conjunctions of linear inequalities φ_1 and φ_2 , we have*

$$\varphi_1 \sqsubseteq^i \varphi_2 \triangleq \varphi_1 \models \varphi_2 = \forall \mathcal{X} : \varphi_1 \Rightarrow \varphi_2.$$

□

Checking $\forall \mathcal{X} : \varphi_1 \Rightarrow \varphi_2$ is simple. We rewrite it to $\neg \exists \mathcal{X} : \varphi_1 \wedge \neg \varphi_2$ and use SMT solvers to check the satisfiability of $\varphi_1 \wedge \neg \varphi_2$. If SMT solvers return “unsat”, we have $\varphi_1 \sqsubseteq^i \varphi_2$. Otherwise, we have $\varphi_1 \not\sqsubseteq^i \varphi_2$.

4.2.2 Equality Testing

To test if two finite conjunctions of linear inequalities are equal, we have the following theorem:

Theorem 4.2.2 (Equality testing for finite conjunctions of linear inequalities). *Given two finite conjunctions of linear inequalities φ_1 and φ_2 , we have*

$$\varphi_1 = \varphi_2 \triangleq \forall \mathcal{X} : \varphi_1 \Leftrightarrow \varphi_2$$

□

Hence, we can use SMT solvers to check if $\varphi_1 \Leftrightarrow \varphi_2$ is valid directly. Instead we can also check $\forall \mathcal{X} : \varphi_1 \Rightarrow \varphi_2$ and $\forall \mathcal{X} : \varphi_2 \Rightarrow \varphi_1$ since $\forall \mathcal{X} : \varphi_1 \Leftrightarrow \varphi_2 = \forall \mathcal{X} : \varphi_1 \Rightarrow \varphi_2 \wedge \varphi_2 \Rightarrow \varphi_1 = \forall \mathcal{X} : \varphi_1 \Rightarrow \varphi_2 \wedge \forall \mathcal{X} : \varphi_2 \Rightarrow \varphi_1$. In the later case, if SMT solvers return “unsat” for both cases, then we have $\varphi_1 = \varphi_2$, otherwise, we have $\varphi_1 \neq \varphi_2$.

4.2.3 Meet

Given two finite conjunctions of linear inequalities, we have the following theorem for the meet (or intersection):

Theorem 4.2.3 (Meet of finite conjunctions of linear inequalities). *Given two finite conjunctions of linear inequalities φ_1 and φ_2 , we have*

$$\varphi_1 \sqcap^i \varphi_2 \triangleq \varphi_1 \wedge \varphi_2$$

□

Hence, we simplify $\varphi_1 \wedge \varphi_2$ by eliminating irrelevant inequalities, let it be φ . We then have $\varphi_1 \sqcap^i \varphi_2 = \varphi$.

4.2.4 Join

Similar to the meet, we have the following theorem for the join (or union) of finite conjunctions of linear inequalities:

Theorem 4.2.4 (Join of finite conjunctions of linear inequalities). *Given two finite conjunctions of linear inequalities φ_1 and φ_2 , we have*

$$\varphi_1 \sqcup^i \varphi_2 \triangleq \varphi_1 \vee \varphi_2$$

□

Unlike the meet, it is often the case that there may not exist a finite conjunction of linear inequalities φ which satisfies $\forall \mathcal{X} : \varphi \Leftrightarrow \varphi_1 \vee \varphi_2$. Instead, we generate a finite conjunction of linear inequalities φ' which satisfies $\forall \mathcal{X} : \varphi_1 \vee \varphi_2 \Rightarrow \varphi'$.

Let φ_1 and φ_2 have the form

$$\begin{aligned}\varphi_1 &= \bigwedge_{i \in [1, l]} a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i \\ \varphi_2 &= \bigwedge_{j \in [1, m]} c_{j1}x_1 + c_{j2}x_2 + \cdots + c_{jn}x_n \leq d_j\end{aligned}$$

We apply convex hull on them and get

$$\begin{aligned}& \exists \sigma_1 : \exists \sigma_2 : \exists y_{11} : \exists y_{21} : \exists y_{12} : \exists y_{22} : \dots : \exists y_{1n} : \exists y_{2n} : \\ & x_1 = y_{11} + y_{21} \wedge x_2 = y_{12} + y_{22} \wedge \dots \wedge x_n = y_{1n} + y_{2n} \\ \wedge & \sigma_1 + \sigma_2 = 1 \wedge \sigma_1 \geq 0 \wedge \sigma_2 \geq 0 \\ \wedge & \bigwedge_{i \in [1, l]} a_{i1}y_{11} + a_{i2}y_{12} + \cdots + a_{in}y_{1n} \leq b_i \sigma_1 \\ \wedge & \bigwedge_{j \in [1, m]} c_{j1}y_{21} + c_{j2}y_{22} + \cdots + c_{jn}y_{2n} \leq d_j \sigma_2\end{aligned} \tag{4.1}$$

Eliminating $\sigma_1, \sigma_2, y_{11}, y_{21}, y_{12}, y_{22}, \dots, y_{1n}, y_{2n}$ will yield a finite conjunction of linear equalities φ which satisfies $\forall \mathcal{X} : \varphi_1 \vee \varphi_2 \Rightarrow \varphi$.

Like other operators in this chapter, SMT solvers can also be used here to solve equation 4.1 thanks to the following famous theorem:

Theorem 4.2.5 (Farkas' Lemma [Sch86]). *Consider a system S of linear inequalities $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n + b_i \leq 0, i \in [1, m]$ over a set of variables x_1, x_2, \dots, x_n . When S is satisfiable, it entails a linear inequality $c_1x_1 + c_2x_2 + \cdots + c_nx_n + d \leq 0$ iff there exist non-negative integers $\lambda_0, \lambda_1, \dots, \lambda_m$ (integers for linear equalities) such that $c_1 = \sum_{i=1}^m \lambda_i a_{i1}, \dots, c_n = \sum_{i=1}^m \lambda_i a_{in}, d = (\sum_{i=1}^m \lambda_i b_i) - \lambda_0$. Moreover, S is unsatisfiable iff the inequality $1 \leq 0$ can be derived. \square*

With Farkas's lemma, the above problem has been transformed to a satisfiability problem over linear integer arithmetic which can be easily solved by SMT solvers. We first rewrite equation 4.1 as:

$$\begin{aligned}
& \exists \sigma_1 : \exists \sigma_2 : \exists y_{11} : \exists y_{21} : \exists y_{12} : \exists y_{22} : \dots : \exists y_{1n} : \exists y_{2n} : \\
& x_1 - y_{11} - y_{21} = 0 \wedge x_2 - y_{12} - y_{22} = 0 \wedge \dots \wedge x_n - y_{1n} - y_{2n} = 0 \\
\wedge & \sigma_1 + \sigma_2 - 1 = 0 \wedge -\sigma_1 \leq 0 \wedge -\sigma_2 \leq 0 \\
\wedge & \bigwedge_{i \in [1, l]} a_{i1}y_{11} + a_{i2}y_{12} + \dots + a_{in}y_{1n} - b_i\sigma_1 \leq 0 \\
\wedge & \bigwedge_{j \in [1, m]} c_{j1}y_{21} + c_{j2}y_{22} + \dots + c_{jn}y_{2n} - d_j\sigma_2 \leq 0
\end{aligned}$$

Applying Farkas' lemma with the template linear inequality $c_1x_1 + \dots + c_nx_n + d \leq 0$ (the coefficients of $\sigma_1, \sigma_2, y_{1i}, y_{2i}$ are all 0), we will get:

$$\begin{aligned}
\exists \lambda_0 : \exists \lambda_1 : \dots : \exists \lambda_{l+m+n+3} : & \lambda_0 \geq 0 \wedge \lambda_{n+2} \geq 0 \wedge \lambda_{n+3} \geq 0 \wedge \dots \\
& \wedge c_1 = \lambda_1 \wedge \dots \wedge c_n = \lambda_n \wedge d = -\lambda_0 - \lambda_{n+1} \\
& \wedge -\lambda_1 + a_{11}\lambda_{n+4} + \dots + a_{l1}\lambda_{l+n+3} = 0 \wedge \dots \\
& \wedge -\lambda_1 + c_{11}\lambda_{m+n+4} + \dots + c_{m1}\lambda_{l+m+n+3} = 0 \wedge \dots \\
& \wedge \lambda_{n+1} - \lambda_{n+2} - b_1\lambda_{n+4} - \dots - b_l\lambda_{l+n+3} = 0 \\
& \wedge \lambda_{n+1} - \lambda_{n+3} - d_1\lambda_{m+n+4} - \dots - d_m\lambda_{l+m+n+3} = 0
\end{aligned}$$

All satisfiable non-zero models for c_1, c_2, \dots, c_n, d will yield a conjunction of linear inequalities φ which satisfies $\forall \mathcal{X} : \varphi_1 \vee \varphi_2 \Rightarrow \varphi$. Normally, there exists infinite models for c_1, c_2, \dots, c_n, d . Hence, we will only yield a finite subset of all models from SMT solvers. One possible way is to check the satisfiability of a finite set of possibilities of c_1, c_2, \dots, c_n, d such as $\forall i : c_i > 0, \forall i : c_i < 0, \forall i \in [1, k] : c_i > 0 \wedge \forall i \in [k+1, n] : c_i < 0$, etc.

Example 4.2.1. Let $\varphi_1 = x_1 \geq 0 \wedge x_2 \geq 0 \wedge x_1 + x_2 \leq 1$ and $\varphi_2 = x_1 \geq 1 \wedge x_2 = 2$. We first create the convex hull of φ_1 and φ_2 , let it be:

$$\begin{aligned}
\exists \sigma_1 : \exists \sigma_2 : \exists y_{11} : \exists y_{21} : \exists y_{12} : \exists y_{22} : & x_1 = y_{11} + y_{21} \wedge x_2 = y_{12} + y_{22} \\
& \wedge \sigma_1 + \sigma_2 = 1 \wedge \sigma_1 \geq 0 \wedge \sigma_2 \geq 0 \\
& \wedge y_{11} \geq 0 \wedge y_{12} \geq 0 \wedge y_{11} + y_{12} \leq \sigma_1 \\
& \wedge y_{21} \geq \sigma_2 \wedge y_{22} = 2\sigma_2
\end{aligned}$$

which then has been rewritten as:

$$\begin{aligned}
\exists \sigma_1 : \exists \sigma_2 : \exists y_{11} : \exists y_{21} : \exists y_{12} : \exists y_{22} : & x_1 - y_{11} - y_{21} = 0 \wedge x_2 - y_{12} - y_{22} = 0 \\
& \wedge \sigma_1 + \sigma_2 - 1 = 0 \wedge -\sigma_1 \leq 0 \wedge -\sigma_2 \leq 0 \\
& \wedge -y_{11} \leq 0 \wedge -y_{12} \leq 0 \wedge y_{11} + y_{12} - \sigma_1 \leq 0 \\
& \wedge -y_{21} + \sigma_2 \leq 0 \wedge y_{22} - 2\sigma_2 = 0
\end{aligned}$$

Applying Farkas' lemma, we will get:

$$\begin{aligned}
\exists \lambda_0 : \exists \lambda_1 : \dots : \exists \lambda_{10} : & \lambda_0 \geq 0 \wedge \lambda_4 \geq 0 \wedge \lambda_5 \geq 0 \wedge \lambda_6 \geq 0 \\
& \wedge \lambda_7 \geq 0 \wedge \lambda_8 \geq 0 \wedge \lambda_9 \geq 0 \\
& \wedge c_1 = \lambda_1 \wedge c_2 = \lambda_2 \wedge d = -\lambda_0 - \lambda_3 \\
& \wedge -\lambda_1 - \lambda_6 + \lambda_8 = 0 \wedge -\lambda_2 - \lambda_7 + \lambda_8 = 0 \\
& \wedge -\lambda_1 - \lambda_9 = 0 \wedge -\lambda_2 + \lambda_{10} = 0 \\
& \wedge \lambda_3 - \lambda_4 - \lambda_8 \wedge \lambda_3 - \lambda_5 + \lambda_9 - 2\lambda_{10} = 0
\end{aligned}$$

By checking the satisfiability of above formula using Z3, we get the following models:

- $c_1 = -1, c_2 = 0, d = 0;$
- $c_1 = 0, c_2 = 1, d = -2;$
- $c_1 = 0, c_2 = -1, d = 0;$
- $c_1 = -1, c_2 = 1, d = -1;$
- $c_1 = -1, c_2 = -1, d = 0.$

which yields $-x_1 \leq 0 \wedge x_2 - 2 \leq 0 \wedge -x_2 \leq 0 \wedge -x_1 + x_2 - 1 \leq 0 \wedge -x_1 - x_2 \leq 0$. Moreover,

we have $\forall x_1 : \forall x_2 : -x_1 \leq 0 \wedge x_2 - 2 \leq 0 \wedge -x_2 \leq 0 \wedge -x_1 + x_2 - 1 \leq 0 \Rightarrow -x_1 - x_2 \leq 0$.

Hence, we have $\varphi_1 \sqcup^i \varphi_2 = -x_1 \leq 0 \wedge x_2 - 2 \leq 0 \wedge -x_2 \leq 0 \wedge -x_1 + x_2 - 1 \leq 0$. \square

4.3 Transfer Functions

We now define transfer functions which are necessary in the program analysis.

4.3.1 Assignment

Given a finite conjunction of linear inequalities φ and an assignment $x_\ell = E$, we consider the following three different kind of assignment transfer functions.

Assignment of a Non Linear Expression

If the expression $E(\mathcal{X})$ is not linear then we assume any value of \mathbb{Z} can be assigned to x_ℓ which means we will know nothing about the value of x_ℓ after the assignment. Therefore we have to eliminate x_ℓ from φ .

One possible method is Fourier-Motzkin elimination method [Sch86]:

1. Rewrite φ into the form $\varphi_N \wedge \varphi_L \wedge \varphi_G$ where φ_N is the conjunction of linear equalities which do not contain x_ℓ , φ_L is the conjunction of linear inequalities which can all be rewritten into the form $x_\ell \leq t_1$ and φ_G is the conjunction of linear inequalities which can all be rewritten into the form $x_\ell \geq t_2$.
2. For each $x_\ell \leq t_1$ in φ_L and each $x_\ell \geq t_2$ in φ_G , we generate a new linear inequality $t_2 \leq t_1$. Let φ_L be a conjunction of m linear inequalities and φ_G be a conjunction of n linear inequalities, we will have $m \times n$ new linear inequalities. Let ψ be the conjunction of these $m \times n$ linear inequalities, return $\varphi_N \wedge \psi$.
3. If either φ_L or φ_G is empty, return φ_N .

Farkas' lemma can be used to eliminating x_ℓ from φ as well by letting $c_\ell = 0$ in the template inequality.

Example 4.3.1. Let $\varphi = x_1 - x_2 \leq 1 \wedge x_2 \leq 1 \wedge x_1 + x_2 \leq 5$. When using Fourier-Motzkin elimination method, the new conjunction of linear inequalities after the assignment $x_2 = x_1 * x_2$ is $\varphi' = x_1 \leq 2 \wedge x_1 \leq 3 = x_1 \leq 2$.

When using Farkas' lemma, we have $\exists \lambda_0 : \exists \lambda_1 : \exists \lambda_2 : \exists \lambda_3 : \lambda_0 \geq 0 \wedge \lambda_1 \geq 0 \wedge \lambda_2 \geq 0 \wedge \lambda_3 \geq 0 \wedge c_1 = \lambda_1 + \lambda_3 \wedge d = -\lambda_0 - \lambda_1 - \lambda_2 - 5\lambda_3 \wedge -\lambda_1 + \lambda_2 + \lambda_3 = 0$. By checking the satisfiability using Z3, we get $c_1 = 2 \wedge d = -6$ which yields $\varphi' = x_1 \leq 3$. \square

Invertible Assignments

When the expression $E(\mathcal{X})$ in the assignment contains x_ℓ , it is called invertible assignment.

Let the assignment has the form

$$x_\ell = a_1x_1 + \dots + a_\ell x_\ell + \dots + a_nx_n + b, \quad a_\ell \neq 0$$

The fact $a_\ell \neq 0$ means we can carry over the knowledge of the previous value of x_ℓ to the new value of x_ℓ . We rewrite the assignment as:

$$x_\ell = [x_i - (a_1x_1 + \dots + a_{\ell-1}x_{\ell-1} + a_{\ell+1}x_{\ell+1} + \dots + a_nx_n + b)]/a_\ell$$

Let $\varphi[x \leftarrow e]$ represent the replacement of x by e in φ , then the assignment transfer function for invertible assignment can be defined as:

$$\begin{aligned} & f[[x_i = a_1x_1 + \dots + a_ix_i + \dots + a_nx_n + b]]\varphi \\ = & \varphi[x_i \leftarrow [x_i - (a_1x_1 + \dots + a_{\ell-1}x_{\ell-1} + a_{\ell+1}x_{\ell+1} + \dots + a_nx_n + b)]/a_\ell] \end{aligned}$$

Example 4.3.2. let $\varphi = x_1 - x_2 \leq 1 \wedge x_2 \leq 1 \wedge x_1 + x_2 \leq 5$. The assignment $x_2 = x_1 + x_2 + 1$ is invertible so that we have $x'_2 = -x_1 + x_2 - 1$. Then the new conjunction of linear inequalities after the assignment is $\varphi' = 2x_1 - x_2 \leq 0 \wedge -x_1 + x_2 \leq 2 \wedge x_2 \leq 3$. \square

Non-invertible Assignments

When the expression $E(\mathcal{X})$ in the assignment does not contain x_ℓ , it is called non-invertible assignment. In this case, we cannot solve φ' in terms of φ so that some information is lost by the assignment. Therefore we should eliminate x_ℓ from φ . This is similar to the

case of assignment of a non linear expression except that the assignment $x_\ell = E$ is added to the result after elimination.

Example 4.3.3. let $\varphi = x_1 - x_2 \leq 1 \wedge x_2 \leq 1 \wedge x_1 + x_2 \leq 5$. The assignment $x_2 = x_1 + 1$ is non-invertible. The elimination of x_2 in φ will get $x_1 \leq 2$ by Fourier Motzkin method or $x_1 \leq 3$ by Farkas' lemma. So the new conjunction of linear inequalities after the assignment is $\varphi' = x_1 \leq 2 \wedge x_1 - x_2 + 1 = 0$ or $\varphi' = x_1 \leq 3 \wedge x_1 - x_2 + 1 = 0$. \square

4.3.2 Test

Given a conjunction of linear inequalities φ and a linear test ϕ , if the test is not in the form of linear equality, we need first reduce it to the form of linear inequality. Consider a generic test $E_1 \bowtie E_2$. A first step is to group the expressions on the left side as $E_1 - E_2 \bowtie 0$. We then do one of the following if \bowtie is not \leq :

- If \bowtie is $=$, the test can be rewritten as:

$$E_1 - E_2 \leq 0 \wedge E_2 - E_1 \leq 0$$

- If \bowtie is $<$, then we can use the test:

$$E_1 - E_2 + 1 \leq 0$$

- If \bowtie is \neq , we calculate the join of two inequalities:

$$E_1 - E_2 + 1 \leq 0 \sqcup^i E_2 - E_1 + 1 \leq 0$$

When we have done the reduction of the test, let it be ψ . We then simply calculate $\varphi'_t = \varphi \sqcap^i \psi$ for the true branch and $\varphi'_f = \varphi \sqcap^i \neg\psi$ for the false branch.

Example 4.3.4. let $\varphi = x_1 - x_2 \leq 1 \wedge x_2 \leq 1 \wedge x_1 + x_2 \leq 5$. A test $x_1 \geq 1 - x_2$ is first rewritten to $-x_1 - x_2 + 1 \leq 0$. Then we have $\varphi'_t = x_1 - x_2 \leq 1 \wedge x_2 \leq 1 \wedge x_1 + x_2 \leq 5 \wedge -x_1 - x_2 \leq -1$ and $\varphi'_f = x_1 - x_2 \leq 1 \wedge x_1 + x_2 \leq 1$. \square

4.4 Widening

Widening is a generic concept that can be used to guarantee convergence of the fixed-point computations in the static analysis. We borrow the standard widening [Hal79] which is universally used in the polyhedra abstract domains and adapt it to the finite conjunctions of linear inequalities.

Definition 4.4.1. Given two finite conjunctions of linear inequalities $\varphi_1 = l_1 \wedge l_2 \wedge \dots \wedge l_m$ and $\varphi_2 = j_1 \wedge j_2 \wedge \dots \wedge j_n$ which satisfy $\varphi_1 \sqsubseteq^i \varphi_2$ where $l_i, j_k, i \in [1, m], k \in [1, n]$ are linear inequalities. The widening of φ_1, φ_2 is:

$$\varphi_1 \nabla^i \varphi_2 \triangleq \psi_1 \wedge \psi_2$$

where

- ψ_1 is a conjunction of linear inequalities where each linear inequality l is in φ_1 which satisfies $\forall \mathcal{X} : \varphi_2 \Rightarrow l$;
- ψ_2 is a conjunction of linear inequalities where each linear inequality j is in φ_2 but not in φ_1 while there exists a linear inequality l in φ_1 which we can replace l by j in φ_1 without changing the concretization of φ_1 . Let $\varphi \setminus l$ represent removing l from φ , then it can be written as $\forall \mathcal{X} : (\varphi_1 \setminus l) \wedge j \Leftrightarrow \varphi_1$.

\square

Both $\forall \mathcal{X} : \varphi_2 \Rightarrow l$ and $\forall \mathcal{X} : (\varphi_1 \setminus l) \wedge j \Leftrightarrow \varphi_1$ can be easily checked by SMT solvers (See 4.2.1 and 4.2.2). Hence we can use SMT solvers to calculate the widening of two finite conjunctions of linear inequalities.

4.5 Example

Let's consider the following program:

```

[P0] x = 2; y = z + 5;
[P1] while [P2] (...) {
    [P3] x = x + 1; y = y + 3;
[P4] }
```

The test is non linear hence it is not taken into account. Initially, P_0^0 is *true* and $P_1^0, P_2^0, P_3^0, P_4^0$ are *false*. Then each assertion is propagated through the analysis of the program:

$$P_0^1 = \textit{true}$$

$$P_1^1 = f[y = z + 5](f[x = 2]P_0^1) = x = 2 \wedge y - z = 5$$

$$P_2^1 = P_1^1 \sqcup^i P_4^1 = P_1^1 \vee \textit{false} = P_1^1$$

$$P_3^1 = P_2^1$$

$$P_4^1 = f[y = y + 3](f[x = x + 1]P_3^1) = x = 3 \wedge y - z = 8$$

The assertion P_0 and P_1 will not be changed, hence we continue with P_2 :

$$P_2^2 = P_1^1 \sqcup^i P_4^1 = 3x - y + z = 1 \wedge 2 \leq x \leq 3$$

$$P_3^2 = P_2^2$$

$$P_4^2 = f[y = y + 3](f[x = x + 1]P_3^2) = 3x - y + z = 1 \wedge 3 \leq x \leq 4$$

When the loop body has been analyzed, we apply widening operation at the loop junction P_2 :

$$P_2^3 = P_2^2 \nabla^i (P_1^1 \sqcup^i P_4^2) = 3x - y + z = 1 \wedge 2 \leq x \leq 3 \nabla^i (x = 2 \wedge y - z = 5 \sqcup^i 3x - y + z = 1 \wedge 3 \leq x \leq 4) = 3x - y + z = 1 \wedge 2 \leq x \leq 3 \nabla^i 3x - y + z = 1 \wedge 2 \leq x \leq 4 = 3x - y + z = 1 \wedge x \geq 2$$

$$P_3^3 = P_2^3$$

$$P_4^3 = f[y = y + 3](f[x = x + 1]P_3^3) = 3x - y + z = 1 \wedge x \geq 3$$

We have $P_1^1 \sqcup^i P_4^3 = 3x - y + z = 1 \wedge x \geq 2$ is equal to P_2^3 so that the program analysis has converged.

The final result is:

```
[P0: true] x = 2; y = z + 5;
[P1: x = 2 ∧ y - z = 5] while [P2: 3x - y + z = 1 ∧ x ≥ 2] (...) {
    [P3: 3x - y + z = 1 ∧ x ≥ 2] x = x + 1; y = y + 3;
[P4: 3x - y + z = 1 ∧ x ≥ 3] }
```

4.6 Implementation

We have implemented our SMT-Based affine equality abstract domain using OCaml. It accepts finite conjunctions of linear inequalities. We translate formulas and SMT queries into SMT-LIB v2 scripts hence all SMT solvers which accept SMT-LIB v2 script as their input can be used in our implementation.

We have evaluated our implementation using different SMT solvers, such as `cvc4` and `z3`. We have also compared our implementation with the same operators in Apron library. Table 4.1 shows the differences between our implementation using `cvc4`, `z3` and Apron when evaluating the program in Sect. 4.5. Each column in the table shows the time usage for calculating the assertion (normally involving one abstract operation) in Sect. 4.5.

Similar to Table 3.1, Table 4.1 shows that there are not too much time differences

	Z3	CVC4	Apron
P_1^1	0.000428s	0.000366s	0.000091s
$P_2^1 = P_1^1 \vee false$	0.000001s	0.000001s	0.000139s
P_4^1	0.000228s	0.000183s	0.000073s
$P_2^2 = P_1^1 \sqcup^i P_4^1$	1.561615s	1.722872s	0.000191s
P_4^2	0.000346s	0.000320s	0.000107s
$P_2^3 = P_2^2 \nabla^i (P_1^1 \sqcup^i P_4^2)$	2.280052s	2.414390s	0.000438s
P_4^3	0.000241s	0.000220s	0.000071s
$P_1^1 \sqcup^i P_4^3$	1.032653s	1.372473s	0.000181s

Table 4.1: Comparing SMT-based linear inequality abstract domain and Polka abstract domain in Apron abstract domain library

between using CVC4 and Z3. But when comparing to the polka abstract domain in Apron abstract domain library, SMT-based linear inequality abstract domain is much slower, especially for computing the join and widening operators, such as P_2^2 and P_2^3 . The similar reasons in Sect. 3.2.5 can be applied here. First, our implementation is not optimal. Second, the join and widening operators need to call the SMT solvers a lot of times. For example, to compute the join in P_2^2 , we need to call the SMT solvers about 12 times. And to compute the join and widening in P_2^3 , we need to call the SMT solvers about 20 times. Moreover, when compare each time used by the SMT solvers between SMT-based affine equality abstract domain and SMT-based linear inequality abstract domain, we can find that the linear inequality abstract domain is slower than the affine equality abstract domain. This implies that the SMT solvers are normally slower in larger formulas. Hence, to improve the performance of our SMT-based linear inequality abstract domain, we may use the same methods in Sect. 3.2.5, such as calling the SMT solvers parallel in the same time. Moreover, the improvement of our SMT-based linear inequality abstract domain should also rely on the improvement of the SMT solvers.

Besides, our SMT-based linear inequality abstract domain often generates less precise results than polka abstract domain in Apron abstract domain library. This is because we only test a few set of possible coefficients for the template inequality hence will not

always yield the optimal results. See Table 4.2 for a simple example of computing the join of $x - 2y \geq 6 \wedge x + 2y \leq 10 \wedge y \geq 0$ and $x - 2y \geq 2 \wedge x + 2y \leq 10 \wedge y \geq 1$.

	Result	Time
Z3	$x - 2y \geq 2 \wedge x + y \geq 5 \wedge x + 2y \leq 10 \wedge y \geq 0$	1.188986s
CVC4	$x - 2y \geq 2 \wedge x + y \geq 5 \wedge x + 2y \leq 10 \wedge y \geq 0$	1.799660s
Apron	$x - 2y \geq 2 \wedge x + 2y \geq 6 \wedge x + 2y \leq 10 \wedge y \geq 0$	0.000237s

Table 4.2: Comparing the join operator in SMT-based linear inequality abstract domain and Polka abstract domain in Apron abstract domain library

4.7 Related Work

To the best of our knowledge, these are the first logical abstract domains which rely on SMT solvers for the computation of transformations and other logical operations. However, the idea of logical abstract interpretation and using SMT solver or other theorem provers to build abstract domains is definitely not new. In [CC92a], The extension of abstract interpretation framework to logic programs has been investigated. The semantics foundations to abstract domains consisting in first order logic formulas in a theory were first given in [CCM10a]. Moreover, in [CCM11], the equivalence between reduced product of abstract domains and the combination of decision procedures has been proved, hence opens up possibilities for interesting synergies between SMT solvers and program analyzers. There also exists several works in computing abstract logical operators based on predicate abstraction, such as in [YRS04, TR12]. All of these works relied on finite domains while ours worked on the infinite domain.

On the other hand, there also exist some static analysis using the SMT solvers and Farkas' lemma. In [CSS03], Farkas' lemma has been used to generate linear invariants. In [BHMR07], Farkas' lemma has been used for the synthesis of invariants expressed in the combined theory of linear arithmetic and uninterpreted function symbols. In [HMM12],

a path sensitive static analyzer has been introduced where SMT solvers have been used for finding the feasible paths during the analysis in order to improve the precision. In [LRR13], SMT solvers and Farkas' lemma have been used to solve constraints in order to generate loop invariants over arrays.

Part III

Binary Tree-based Abstract Domains

Chapter 5

Branch Condition Path Abstraction

In this chapter, we introduce branch condition graphs $\mathbb{G}^b[[C]]$ which can be viewed as further abstractions of the control flow graphs $\mathbb{G}[[C]]$ of command C (see Sect. 1.2.5). We then define the branch condition path semantics $\mathcal{G}^b[[\mathbb{G}^b[[C]]]]$ as an abstract interpretation α^b of the action path semantics $\mathcal{G}^a[[C]]$ of the control flow graph $\mathbb{G}[[C]]$ of command C .

5.1 Branch Condition Graph

A *branch condition* is the test B occurring in a command “if (B) $\{C_1\}$ else $\{C_2\}$ ” while a *loop condition* is the test B occurring in a command “while (B) $\{C\}$ ”. A *branch condition graph* (BCG) of a program is a directed acyclic graph, in which each node corresponds to a branch condition occurring in the program and has two outgoing edges representing its true and false branches. An edge from node A to node B means that the branch condition corresponding to node B occurs after the branch condition corresponding to node A in the program and there are no other branch conditions occurring between them. A trace from the entry point to the exit point of a BCG is called *branch condition path*. We use B to denote the true branch while $\neg B$ denotes the false branch.

Example 5.1.1. Consider the following program fragment and its branch condition graph on the left:

```

while(i <= m) {
  if(x < y) x++;
  else y++;
  if(p > 0)
    if(q > 0) r = p + q;
    else r = p - q;
  else
    if(q > 0) r = q - p;
    else r = -(p + q);
  i++;
}

```

```

graph TD
  Start(( )) --> N1[x < y]
  N1 --> N2[p > 0]
  N1 --> N2
  N2 --> N3[q > 0]
  N2 --> N4[q > 0]
  N3 --> End(( ))
  N4 --> End

```

We can generate its branch condition paths as below:

- $(x < y) \cdot (p > 0) \cdot (q > 0),$
- $(x < y) \cdot (p > 0) \cdot \neg(q > 0),$
- $(x < y) \cdot \neg(p > 0) \cdot (q > 0),$
- $(x < y) \cdot \neg(p > 0) \cdot \neg(q > 0),$
- $\neg(x < y) \cdot (p > 0) \cdot (q > 0),$
- $\neg(x < y) \cdot (p > 0) \cdot \neg(q > 0),$
- $\neg(x < y) \cdot \neg(p > 0) \cdot (q > 0),$
- $\neg(x < y) \cdot \neg(p > 0) \cdot \neg(q > 0).$

□

The branch condition graph $\mathbb{G}^b[[C]]$, like the CFG, can be defined in the structural induction on the syntax of the command C :

$$\begin{aligned}
\mathbb{G}^b[[\text{skip}]] &\triangleq \circ \longrightarrow \circ \\
\mathbb{G}^b[[x := E]] &\triangleq \circ \longrightarrow \circ \\
\mathbb{G}^b[[C_1; C_2]] &\triangleq \text{let } \mathbb{G}^b[[C_1]] = \circ \longrightarrow \boxed{C_1^b} \longrightarrow \circ \text{ and} \\
&\quad \mathbb{G}^b[[C_2]] = \circ \longrightarrow \boxed{C_2^b} \longrightarrow \circ \text{ in} \\
&\quad \circ \longrightarrow \boxed{C_1^b} \longrightarrow \boxed{C_2^b} \longrightarrow \circ \\
\mathbb{G}^b[[\text{if } (B) \{C_1\} \text{ else } \{C_2\}]] &\triangleq \text{let } \mathbb{G}^b[[C_1]] = \circ \longrightarrow \boxed{C_1^b} \longrightarrow \circ \text{ and} \\
&\quad \mathbb{G}^b[[C_2]] = \circ \longrightarrow \boxed{C_2^b} \longrightarrow \circ \text{ in} \\
&\quad \circ \longrightarrow \boxed{B} \begin{cases} \text{tt} \longrightarrow \boxed{C_1^b} \\ \text{ff} \longrightarrow \boxed{C_2^b} \end{cases} \longrightarrow \circ \\
\mathbb{G}^b[[\text{while } (B) \{C\}]] &\triangleq \text{let } \mathbb{G}^b[[C]] = \circ \longrightarrow \boxed{C^b} \longrightarrow \circ \text{ in} \\
&\quad \circ \longrightarrow \boxed{C^b} \longrightarrow \circ
\end{aligned}$$

Note that the concatenation of $\circ \longrightarrow \circ$ and $\circ \longrightarrow \circ$ is still $\circ \longrightarrow \circ$.

5.2 Branch Condition Path Abstraction

We abstract finite action paths $A_1 \cdot A_2 \cdot \dots \cdot A_n, n \geq 0$ by the finite branch condition path $A_1^b \cdot A_2^b \cdot \dots \cdot A_m^b, m \leq n$ where $A_1^b = A_p, A_2^b = A_q, \dots, A_m^b = A_r, 1 \leq p < q < \dots < r \leq n$ are distinct branch conditions. The branch condition path is empty ε when there are no branch conditions occurring in the action path. We say that two branch conditions

A_1^b, A_2^b are equal if and only if A_1^b and A_2^b occur at the same program point. Moreover, each branch condition in the branch condition path must be the last occurrence in the action path being abstracted, that is, if A_i^b is a branch condition in the branch condition path $A_1^b \cdot A_2^b \cdot \dots \cdot A_m^b$ abstracting the action path $A_1 \cdot A_2 \cdot \dots \cdot A_n$ where $A_i^b = A_j$, then $\forall k : j < k \leq n, A_i^b \neq A_k$. Note that we only consider finite action paths hence safety properties.

5.2.1 Condition Path Abstraction

An action can be either a skip or an assignment or a condition. The condition path abstraction collects the set of finite sequences of conditions performed along the action path $\underline{\pi}$ and ignores any skip and assignment in $\underline{\pi}$.

Given an action path $\underline{\pi}$, $\alpha^c(\underline{\pi})$ collects the sequence of conditions in the action path, which may be empty ε when there are no conditions occurred in the action path, by the following induction rules:

$$\begin{aligned} \alpha^c(\text{skip}) &\triangleq \varepsilon & \alpha^c(B) &\triangleq B \\ \alpha^c(x = E) &\triangleq \varepsilon & \alpha^c(\neg B) &\triangleq \neg B \\ \alpha^c(\underline{\pi}_1 \cdot \underline{\pi}_2) &\triangleq \alpha^c(\underline{\pi}_1) \cdot \alpha^c(\underline{\pi}_2) \end{aligned}$$

Note that $\varepsilon \cdot \underline{\pi}_c = \underline{\pi}_c \cdot \varepsilon = \underline{\pi}_c$.

Let \mathbb{A}^C be the set of conditions and $(\mathbb{A}^C)^*$ be the set of finite, possibly empty, condition paths. Given a set of action paths \mathcal{A} , $\alpha^c(\mathcal{A})$ collects the sequences of conditions in the action paths \mathcal{A} :

$$\begin{aligned} \alpha^c &\in \wp(\mathbb{A}^*) \mapsto \wp((\mathbb{A}^C)^*) \\ \alpha^c(\mathcal{A}) &\triangleq \{\alpha^c(\underline{\pi}) \mid \underline{\pi} \in \mathcal{A}\} \end{aligned} \tag{5.1}$$

It's easy to see that α^c preserves arbitrary unions and intersections hence is \subseteq -

increasing. By defining $\gamma^c(\mathcal{C}) \triangleq \{\underline{\pi} \mid \alpha^c(\underline{\pi}) \in \mathcal{C}\}$, we will have:

Corollary 5.2.1.

$$(\wp(\mathbb{A}^*), \subseteq) \xrightleftharpoons[\alpha^c]{\gamma^c} (\wp((\mathbb{A}^C)^*), \subseteq) \quad (5.2)$$

Proof. By Theorem 1.2.1 where h is α^c . □

Furthermore, we can construct a concretization γ^c in an inductive manner. Given a condition path $\underline{\pi}_c$, $\gamma^c(\underline{\pi}_c)$ collects all possible action paths whose condition path abstractions are $\underline{\pi}_c$ using following induction rules:

$$\begin{aligned} \gamma^c(\mathbb{A}^c) &\triangleq (\mathbb{A} \setminus \mathbb{A}^C)^* \cdot \mathbb{A}^c \cdot (\mathbb{A} \setminus \mathbb{A}^C)^* \\ \gamma^c(\underline{\pi}_{c_1} \cdot \underline{\pi}_{c_2}) &\triangleq \{a \cdot b \mid a \in \gamma^c(\underline{\pi}_{c_1}) \wedge b \in \gamma^c(\underline{\pi}_{c_2})\} \end{aligned}$$

Then given a set of condition paths \mathcal{C} , $\gamma^c(\mathcal{C})$ collects all possible action paths:

$$\begin{aligned} \gamma^c &\in \wp((\mathbb{A}^C)^*) \mapsto \wp(\mathbb{A}^*) \\ \gamma^c(\mathcal{C}) &\triangleq \bigcup \{\gamma^c(\underline{\pi}_c) \mid \underline{\pi}_c \in \mathcal{C}\} \end{aligned} \quad (5.3)$$

5.2.2 Loop Condition Elimination

Given a finite condition path $\underline{\pi}_c$, $\alpha^d(\underline{\pi}_c)$ collects the finite sequence of branch conditions (with duplications) by eliminating all loop conditions in $\underline{\pi}_c$. This sequence may be empty ε when there are no branch conditions occurred in $\underline{\pi}_c$. Let \mathbb{A}^B be the set of branch conditions and \mathbb{A}^L be the set of loop conditions, thus $\mathbb{A}^C \triangleq \mathbb{A}^B \cup \mathbb{A}^L$ and $\mathbb{A}^B \cap \mathbb{A}^L = \emptyset$. Note that we distinguish those conditions by the program points where they occur, not by themselves.

For all $A^b \in \mathbb{A}^B$ and $A^l \in \mathbb{A}^L$, we have

$$\alpha^d(A^b) \triangleq A^b \quad \text{and} \quad \alpha^d(A^l) \triangleq \varepsilon.$$

Then given two condition paths $\underline{\pi}_{c_1}$ and $\underline{\pi}_{c_2}$, we have

$$\alpha^d(\underline{\pi}_{c_1} \cdot \underline{\pi}_{c_2}) \triangleq \alpha^d(\underline{\pi}_{c_1}) \cdot \alpha^d(\underline{\pi}_{c_2}).$$

Note that $\varepsilon \cdot \underline{\pi}_d = \underline{\pi}_d \cdot \varepsilon = \underline{\pi}_d$.

Let $(\mathbb{A}^B)^*$ be the set of finite, possibly empty, sequences of branch conditions. Given a set of condition paths \mathcal{C} , $\alpha^d(\mathcal{C})$ collects the sequences of branch conditions (with duplications) from the condition paths \mathcal{C} :

$$\begin{aligned} \alpha^d &\in \wp((\mathbb{A}^C)^*) \mapsto \wp((\mathbb{A}^B)^*) \\ \alpha^d(\mathcal{C}) &\triangleq \{\alpha^d(\underline{\pi}_c) \mid \underline{\pi}_c \in \mathcal{C}\} \end{aligned} \quad (5.4)$$

It follows that α^d preserves both arbitrary unions and intersections, hence it is \subseteq -increasing. By defining $\gamma^d(\mathcal{D}) \triangleq \{\underline{\pi}_c \mid \alpha^d(\underline{\pi}_c) \in \mathcal{D}\}$, we will have:

Corollary 5.2.2.

$$(\wp((\mathbb{A}^C)^*), \subseteq) \xleftrightarrow[\alpha^d]{\gamma^d} (\wp((\mathbb{A}^B)^*), \subseteq) \quad (5.5)$$

Proof. By Theorem 1.2.1 where h is α^d . □

Furthermore, we can construct a concretization γ^d in an inductive manner:

$$\begin{aligned} \gamma^d(\mathbb{A}^b) &\triangleq (\mathbb{A}^L)^* \cdot \mathbb{A}^b \cdot (\mathbb{A}^L)^* \\ \gamma^d(\underline{\pi}_{d_1} \cdot \underline{\pi}_{d_2}) &\triangleq \{a \cdot b \mid a \in \gamma^d(\underline{\pi}_{d_1}) \wedge b \in \gamma^d(\underline{\pi}_{d_2})\} \end{aligned}$$

Then given a set of sequences of branch conditions (with duplications) \mathcal{D} , $\gamma^d(\mathcal{D})$ collects all possible condition paths:

$$\begin{aligned} \gamma^d &\in \wp((\mathbb{A}^B)^*) \mapsto \wp((\mathbb{A}^C)^*) \\ \gamma^d(\mathcal{D}) &\triangleq \bigcup \{\gamma^d(\underline{\pi}_d) \mid \underline{\pi}_d \in \mathcal{D}\} \end{aligned} \quad (5.6)$$

5.2.3 Duplication Elimination

The branch condition paths are the sequences of branch conditions without duplications. In this section, we introduce the abstraction function that eliminates duplications in any sequence of branch conditions.

We first define two functions that are used in the abstraction function α^ℓ . Given a sequence seq and an element d of seq , $erase(seq, d)$ eliminates all elements in seq that is equal to d :

$$\begin{aligned} erase(d_1 d_2 d_3 \dots d_n, d) &\triangleq \text{if } d_1 == d \text{ then } erase(d_2 d_3 \dots d_n, d) \\ &\quad \text{else } d_1 \cdot erase(d_2 d_3 \dots d_n, d) \end{aligned}$$

Note that $erase(seq, d)$ may return empty ε . Then $fold(seq)$ eliminates the duplications of each element in seq starting from the last element:

$$\begin{aligned} fold(d_1 d_2 \dots d_n) &\triangleq \text{if } d_1 d_2 \dots d_n == \varepsilon \text{ then } \varepsilon \\ &\quad \text{else } fold(erase(d_1 d_2 \dots d_{n-1}, d_n)) \cdot d_n \end{aligned}$$

Hence, given a sequence of branch conditions $\underline{\pi}_d$, $\alpha^\ell(\underline{\pi}_d) = fold(\underline{\pi}_d)$ eliminates duplications of each branch condition while keeping its last occurrence in $\underline{\pi}_d$.

Let \mathbb{D} be the set of sequences of branch conditions that have duplications. Given a set of sequences of branch conditions \mathcal{D} , $\alpha^\ell(\mathcal{D})$ collects branch condition paths (sequences of branch conditions without duplications):

$$\begin{aligned} \alpha^\ell &\in \wp((\mathbb{A}^B)^*) \mapsto \wp((\mathbb{A}^B)^* \setminus \mathbb{D}) \\ \alpha^\ell(\mathcal{D}) &\triangleq \{\alpha^\ell(\underline{\pi}_d) \mid \underline{\pi}_d \in \mathcal{D}\} \end{aligned} \tag{5.7}$$

It appears that α^ℓ preserves both arbitrary unions and intersections, hence is \subseteq -increasing. By defining $\gamma^\ell(\mathcal{B}) \triangleq \{\underline{\pi}_d \mid \alpha^\ell(\underline{\pi}_d) \in \mathcal{B}\}$, we will have:

Corollary 5.2.3.

$$(\wp((\mathbb{A}^B)^*), \subseteq) \xrightleftharpoons[\alpha^\ell]{\gamma^\ell} (\wp((\mathbb{A}^B)^* \setminus \mathbb{D}), \subseteq) \quad (5.8)$$

Proof. By Theorem 1.2.1 where h is α^ℓ . □

Furthermore, we can define a function $unfold(seq)$ that reverse $fold(seq)$:

$$unfold(d_1 d_2 \dots d_n) \triangleq \{d_1, d_2, \dots, d_n\}^* \cdot d_1 \cdot \{d_2, \dots, d_n\}^* \cdot d_2 \cdot \dots \cdot \{d_n\}^* \cdot d_n$$

Let $\gamma^\ell(\underline{\pi}_b) = unfold(\underline{\pi}_b)$. Then, given a set of branch condition paths \mathcal{B} , we can construct $\gamma^\ell(\mathcal{B})$ as:

$$\begin{aligned} \gamma^\ell &\in \wp((\mathbb{A}^B)^* \setminus \mathbb{D}) \mapsto \wp((\mathbb{A}^B)^*) \\ \gamma^\ell(\mathcal{B}) &\triangleq \bigcup \{\gamma^\ell(\underline{\pi}_b) \mid \underline{\pi}_b \in \mathcal{B}\} \end{aligned} \quad (5.9)$$

5.2.4 Branch Condition Path Abstraction

The branch condition path abstraction $\alpha^b[\mathcal{A}]$ collects the branch condition paths, which is the set of sequences of branch conditions with no duplications along the action paths in \mathcal{A} . It can be defined by the composition of $\alpha^c, \alpha^d, \alpha^\ell$ defined in the previous sections as:

$$\begin{aligned} \alpha^b &\in \wp(\mathbb{A}^*) \mapsto \wp((\mathbb{A}^B)^* \setminus \mathbb{D}) \\ \alpha^b(\mathcal{A}) &\triangleq \alpha^\ell \circ \alpha^d \circ \alpha^c(\mathcal{A}) \end{aligned} \quad (5.10)$$

Respectively, the concretization function $\gamma^b(\mathcal{B})$ can be defined by the composition of $\gamma^c, \gamma^d, \gamma^\ell$ as:

$$\begin{aligned} \gamma^b &\in \wp((\mathbb{A}^B)^* \setminus \mathbb{D}) \mapsto \wp(\mathbb{A}^*) \\ \gamma^b(\mathcal{B}) &\triangleq \gamma^c \circ \gamma^d \circ \gamma^\ell(\mathcal{B}) \end{aligned} \quad (5.11)$$

It follows that α^b and γ^b form a Galois connection:

$$(\wp(\mathbb{A}^*), \subseteq) \xrightleftharpoons[\alpha^b]{\gamma^b} (\wp((\mathbb{A}^B)^* \setminus \mathbb{D}), \subseteq) \quad (5.12)$$

Proof. The composition of Galois connections is still a Galois connection. For all $\mathcal{A} \in \wp(\mathbb{A}^*)$ and $\mathcal{B} \in \wp((\mathbb{A}^B)^* \setminus \mathbb{D})$,

$$\begin{aligned} & \alpha^b(\mathcal{A}) \subseteq \mathcal{B} \\ \iff & \alpha^\ell \circ \alpha^d \circ \alpha^c(\mathcal{A}) \subseteq \mathcal{B} && \text{\{definition of } \alpha^b\}} \\ \iff & \alpha^d \circ \alpha^c(\mathcal{A}) \subseteq \gamma^\ell(\mathcal{B}) && \text{\{by } (\wp((\mathbb{A}^B)^*), \subseteq) \xrightleftharpoons[\alpha^\ell]{\gamma^\ell} (\wp((\mathbb{A}^B)^* \setminus \mathbb{D}), \subseteq)\}} \\ \iff & \alpha^c(\mathcal{A}) \subseteq \gamma^d \circ \gamma^\ell(\mathcal{B}) && \text{\{by } (\wp((\mathbb{A}^C)^*), \subseteq) \xrightleftharpoons[\alpha^d]{\gamma^d} (\wp((\mathbb{A}^B)^*), \subseteq)\}} \\ \iff & \mathcal{A} \subseteq \gamma^c \circ \gamma^d \circ \gamma^\ell(\mathcal{B}) && \text{\{by } (\wp(\mathbb{A}^*), \subseteq) \xrightleftharpoons[\alpha^c]{\gamma^c} (\wp((\mathbb{A}^C)^*), \subseteq)\}} \\ \iff & \mathcal{A} \subseteq \gamma^b(\mathcal{B}) && \text{\{definition of } \gamma^b\}} \end{aligned}$$

□

Chapter 6

Binary Decision Tree Abstract

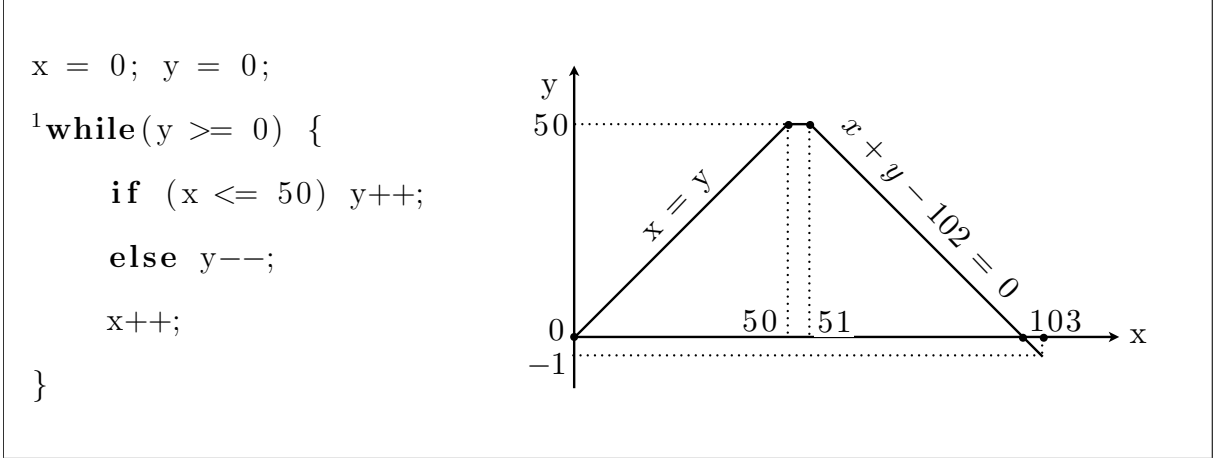
Domain Functor

In abstract interpretation, normally the more disjunctions we allow to encode the program properties, the more precise result we will often have; in the meantime, the combinatorial explosion of disjunctions will also cause much less efficiency. In this chapter, we introduce the binary decision tree abstract domain functor to represent and manipulate invariants in the form of binary decision trees. The abstract property will be represented by the disjunction of leaves which are separated by the values of binary decisions, i.e., boolean tests, which will be organized in the decision nodes of the binary decision trees. This binary decision tree abstract domain functor may provide a flexible way of adjusting the cost/precision ratio in path-dependent static analysis.

6.1 Introduction

Let us first consider the following example which is modified from the one in [GR07]:

Example 6.1.1. A motivating example.



We know that the strongest invariant at program point ¹ is $(0 \leq x \leq 50 \wedge x = y) \vee (51 \leq x \leq 103 \wedge x + y - 102 = 0)$. When we use the APRON numerical abstract domain library [JM09] to generate the invariant at program point ¹, we get $x \geq 0 \wedge y \geq -1$ with the box (interval) abstract domain and $y \geq -1 \wedge x - y \geq 0 \wedge x + 52y \geq 0$ with the polka (convex polyhedra) abstract domain. Both analyses are very imprecise compared to the strongest one. This is because the true and false branches of “if ($x \leq 50$)” have different behaviors and those abstract domains do not consider them separately. \square

Hence, we propose the binary decision tree abstract domain functor that takes those branches into consideration.

Given the trace semantics $\mathcal{S}^t[[P]]$ of a program P , $\alpha^b \circ \alpha^a(\mathcal{S}^t[[P]])$ abstracts $\mathcal{S}^t[[P]]$ into a finite set \mathcal{B} of branch condition paths where $|\mathcal{B}| = N$. Then for each $\pi_b \in \mathcal{B}$, we have $\gamma^a \circ \gamma^b(\pi_b) \cap \mathcal{S}^t[[P]] \subseteq \mathcal{S}^t[[P]]$ and $\bigcup_{i \leq N} (\gamma^a \circ \gamma^b(\pi_{b_i}) \cap \mathcal{S}^t[[P]]) = \mathcal{S}^t[[P]]$. Moreover, for all pairs $(\pi_{b_1}, \pi_{b_2}) \in \mathcal{B} \times \mathcal{B}$, we have $(\gamma^a \circ \gamma^b(\pi_{b_1}) \cap \mathcal{S}^t[[P]]) \cap (\gamma^a \circ \gamma^b(\pi_{b_2}) \cap \mathcal{S}^t[[P]]) = \emptyset$. Hence, the set \mathcal{B} of branch condition paths defines a partitioning of the trace semantics $\mathcal{S}^t[[P]]$ of a program P . If we can generate the invariants for each program point only using the information of one partition of the trace semantics, then for each program point, we will get a finite set of invariants. It follows that the disjunction of such set of invariants forms the invariant of that program point. Hence, we encapsulate the set of branch condition

paths into the decision nodes of a binary decision tree where each top-down path (without leaf) of the binary decision tree represents a branch condition path, and store in each leaf nodes the invariant generated from the information of the subset of the trace semantics defined by the corresponding branch condition path.

We denote the binary decision tree in the parenthesized form

$$\llbracket B_1 : \llbracket B_2 : \langle P_1 \rangle, \langle P_2 \rangle \rrbracket, \llbracket B_3 : \langle P_3 \rangle, \langle P_4 \rangle \rrbracket \rrbracket$$

where B_1, B_2, B_3 are decisions (branch conditions) and P_1, P_2, P_3, P_4 are invariants. It encodes the fact that either if B_1 and B_2 are both true then P_1 holds, or if B_1 is true and B_2 is false then P_2 holds, or if B_1 is false and B_3 is true then P_3 holds, or if B_1 and B_3 are both false then P_4 holds. The parenthesized representation of binary trees uses $\langle \dots \rangle$ for leaves and $\llbracket B : t_l, t_r \rrbracket$ for the decision B and t_l (resp. t_r) represents its left subtree (resp. right subtree). In first order logic, the above binary decision tree would be written as $(B_1 \wedge B_2 \wedge P_1) \vee (B_1 \wedge \neg B_2 \wedge P_2) \vee (\neg B_1 \wedge B_3 \wedge P_3) \vee (\neg B_1 \wedge \neg B_3 \wedge P_4)$ with an implicit universal quantification over free variables.

Let $D(\mathcal{B})$ denote the set of all branch conditions appearing in \mathcal{B} . Let $\beta = B$ or $\neg B$ and $\mathcal{B}_{\setminus \beta}$ denote the removal of β and all branch conditions appearing before in each branch condition path in \mathcal{B} , then we define the binary decision tree as:

Definition 6.1.1 (Binary Decision Tree). A binary decision tree $t \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$ over the set \mathcal{B} of branch condition paths (with concretization $\gamma^a \circ \gamma^b$) and the leaf abstract domain \mathbb{D}_ℓ (with concretization γ_ℓ) is either $\langle p \rangle$ with p is an element of \mathbb{D}_ℓ and \mathcal{B} is empty or $\llbracket B : t_t, t_f \rrbracket$ where $B \in D(\mathcal{B})$ is the first element of all branch condition paths $\underline{\pi}_b \in \mathcal{B}$ and (t_t, t_f) are the left and right subtree of t represent its true and false branch such that $t_t, t_f \in \mathbb{T}(\mathcal{B}_{\setminus \beta}, \mathbb{D}_\ell)$. □

Example 6.1.2. In Example 6.1.1, the binary decision tree at program point ¹ will be $t = \llbracket x \leq 50 : (0 \leq x \leq 50 \wedge x = y), (51 \leq x \leq 103 \wedge x + y - 102 = 0) \rrbracket$. \square

From theorem 6.1.1, we can also define the binary decision tree in the fixpoint form:

Theorem 6.1.1. *The following least fixpoint form also defines the binary decision trees*

$$\mathbb{T}(\mathcal{B}, \mathbb{D}_\ell) = \text{lfp}^\subseteq F(\mathcal{B}, \mathbb{D}_\ell)$$

where

$$\begin{aligned} F(\mathcal{B}, \mathbb{D}_\ell)X \triangleq & \{ \langle p \rangle \mid p \in \mathbb{D}_\ell \} \\ & \cup \{ \llbracket B : t_t, t_f \rrbracket \mid B \in D(\mathcal{B}) \wedge t_t, t_f \in X \wedge \\ & \qquad \qquad \qquad X \subseteq \mathbb{T}(\mathcal{B}_{\setminus \beta}, \mathbb{D}_\ell) \} \end{aligned}$$

Proof. The proof is straight forward from Definition 6.1.1. \square

Let ρ be the concrete environment assigning concrete values $\rho(x)$ to variables x and $\llbracket e \rrbracket \rho$ for the concrete value of the expression e in the concrete environment ρ , we can then define the concretization of the binary decision tree as

Definition 6.1.2. The concretization of a binary decision tree γ_t is either

$$\gamma_t(\langle p \rangle) \triangleq \gamma_\ell(p)$$

when the binary decision tree can be reduced to a leaf or

$$\begin{aligned} \gamma_t(\llbracket B : t_t, t_f \rrbracket) \triangleq & \{ \rho \mid \llbracket B \rrbracket \rho = \text{true} \implies \rho \in \gamma_t(t_t) \wedge \\ & \llbracket B \rrbracket \rho = \text{false} \implies \rho \in \gamma_t(t_f) \} \end{aligned}$$

when the binary decision tree is rooted at a decision node. \square

Furthermore, we would like $\gamma_t(t_t) \neq \gamma_t(t_f)$ in any circumstances. So we revise the definition of binary decision tree as below:

Definition 6.1.3. The revised Binary Decision Tree $t \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$ with its concretization γ_t are

$$\begin{aligned}
\langle t, \gamma_t(t) \rangle &\triangleq \{ \langle \langle p \rangle, \gamma_\ell(p) \rangle \mid p \in \mathbb{D}_\ell \} \\
&\vee \{ \langle \llbracket B : t_t, t_f \rrbracket, \rho \rangle \mid B \in D(\mathcal{B}) \\
&\quad \wedge t_t, t_f \in \mathbb{T}(\mathcal{B}_{\setminus \beta}, \mathbb{D}_\ell) \\
&\quad \wedge \llbracket B \rrbracket \rho = \text{true} \implies \rho \in \gamma_t(t_t) \\
&\quad \wedge \llbracket B \rrbracket \rho = \text{false} \implies \rho \in \gamma_t(t_f) \\
&\quad \wedge \gamma_t(t_t) \neq \gamma_t(t_f) \}
\end{aligned}$$

□

Given $t_1, t_2 \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$, we say that $t_1 \equiv_t t_2$ if and only if $\gamma_t(t_1) = \gamma_t(t_2)$. Let $\mathbb{T}(\mathcal{B}, \mathbb{D}_\ell) \setminus \equiv_t$ be the quotient by the equivalence relation \equiv_t . The binary decision tree abstract domain functor is defined as:

Definition 6.1.4. A binary decision tree abstract domain functor is a tuple

$$\langle \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell) \setminus \equiv_t, \sqsubseteq_t, \perp_t, \top_t, \sqcup_t, \sqcap_t, \nabla_t, \Delta_t \rangle$$

on two parameters, a set \mathcal{B} of branch condition paths and a leaf abstract domain \mathbb{D}_ℓ where

$$\begin{aligned}
P, Q, \dots &\in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell) \setminus \equiv_t && \text{abstract properties} \\
\sqsubseteq_t &\in \mathbb{T} \times \mathbb{T} \rightarrow \{\text{false}, \text{true}\} && \text{abstract partial order} \\
\perp_t, \top_t &\in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell) && \text{infimum, supremum} \\
&&& (\forall P \in \mathbb{T} : \perp_t \sqsubseteq_t P \sqsubseteq_t \top_t) && (6.1) \\
\sqcup_t, \sqcap_t &\in \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T} && \text{abstract join, meet} \\
\nabla_t, \Delta_t &\in \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T} && \text{abstract widening, narrowing} && \square
\end{aligned}$$

The set \mathcal{B} of branch condition paths is built by the syntactic analysis from the control flow of the program. Hence the structure of the binary decision tree is finite and does not change during the data flow analysis. The static analyzer designer should allow to change the maximal length of branch condition paths in \mathcal{B} so as to be able to adjust the cost/precision ratio of the analysis. The leaf abstract domain \mathbb{D}_ℓ for the leaves could be any numerical or symbolic abstract domains such as intervals, octagons, polyhedra, array domains, etc., or the logical abstract domains we defined in Chapter 3 and Chapter 4, or even the reduced product of two or more of those abstract domains. A list of available abstract domains that can be used at the leaves would be another option of the static analyzer designer. We can use any of these options to build a particular instance of the binary decision tree abstract functor. The advantage of this modular approach is that we can change those options to adjust the cost/precision ratio without having to change the structure of the analyzer.

6.2 Binary Operations

In this section, we introduce several abstract binary operations, such as abstract partial order, join and meet, for the binary decision tree abstract domain functor.

6.2.1 Inclusion and Equality

Given two binary decision trees $t_1, t_2 \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell) \setminus \{\perp_t, \top_t\}$, we can check $t_1 \sqsubseteq_t t_2$ by comparing each pair (ℓ_1, ℓ_2) of leaves in (t_1, t_2) where ℓ_1 and ℓ_2 are defined by the same branch condition path $\pi_b \in \mathcal{B}$. If each pair (ℓ_1, ℓ_2) satisfies $\ell_1 \sqsubseteq_\ell \ell_2$, we can conclude that $t_1 \sqsubseteq_t t_2$; otherwise, we have $t_1 \not\sqsubseteq_t t_2$.

```

\* To check whether  $t1 \sqsubseteq_t t2$ . *\
include(t1, t2 : binary decision trees)
{
    if (t1 == (l1)) && t2 == (l2)) then return  $t1 \sqsubseteq_\ell t2$ ;

    let t1 = [[B: t1l, t1r]] and t2 = [[B: t2l, t2r]];
    return include(t1l, t2l) & include(t1r, t2r);
}

```

Example 6.2.1. We have $\llbracket x \leq 50 : (x = 0 \wedge y = 0), (\perp_\ell) \rrbracket \sqsubseteq \llbracket x \leq 50 : (0 \leq x \leq 1 \wedge x = y), (\perp_\ell) \rrbracket$ and $\llbracket x \leq 50 : (x = 0 \wedge y = 0), (\perp_\ell) \rrbracket \not\sqsubseteq \llbracket x \leq 50 : (x = 1 \wedge y = 1), (\perp_\ell) \rrbracket$.

□

The equality of t_1 and t_2 can be tested by the fact $t_1 =_t t_2 \stackrel{\Delta}{=} t_1 \sqsubseteq_t t_2 \wedge t_2 \sqsubseteq_t t_1$. When the leaf abstract domain \mathbb{D}_ℓ has $=_\ell$, we can also check the equality for each pair (ℓ_1, ℓ_2) of leaves in (t_1, t_2) where ℓ_1 and ℓ_2 are defined by the same branch condition path $\underline{\pi}_b \in \mathcal{B}$.

6.2.2 Meet and Join

Given two binary decision trees $t_1, t_2 \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$, the meet $t = t_1 \sqcap_t t_2$ can be computed using the meet \sqcap_ℓ in the leaf abstract domain \mathbb{D}_ℓ . Let ℓ_1, ℓ_2 be leaves of t_1, t_2 respectively, where the same branch condition path $\underline{\pi}_b \in \mathcal{B}$ leads to ℓ_1 and ℓ_2 , then $\ell = \ell_1 \sqcap_\ell \ell_2$ is the leaf of t led by the same branch condition path $\underline{\pi}_b \in \mathcal{B}$. After computing each leaf $\ell = \ell_1 \sqcap_\ell \ell_2$ in t , we then get $t = t_1 \sqcap_t t_2$.

```

meet(t1, t2 : binary decision trees)
{
  if (t1 == (l1)) && t2 == (l2)) then return t1  $\sqcap_\ell$  t2;

  let t1 = [[B: t1l, t1r]] and t2 = [[B: t2l, t2r]];
  return [[B: meet(t1l, t2l), meet(t1r, t2r)]];
}

```

Similar to the meet, we can compute the join $t = t_1 \sqcup_\ell t_2$ using the join \sqcup_ℓ in the leaf abstract domain \mathbb{D}_ℓ . Instead of computing the join $\ell_1 \sqcup_\ell \ell_2$ for each pair (ℓ_1, ℓ_2) of leaves in (t_1, t_2) where ℓ_1 and ℓ_2 are led by the same branch condition path $\underline{\pi}_b \in \mathcal{B}$, we also use the branch conditions in $\underline{\pi}_b$ as bound to prevent precision loss. Let $\underline{\pi}_b = \beta_1 \cdot \beta_2 \cdot \dots \cdot \beta_n$ where $\beta_i = B_i$ or $\neg B_i, i = 1, \dots, n$, we have $\ell = (\ell_1 \sqcup_\ell \ell_2) \sqcap_\ell \mathbb{D}_\ell(\beta_1) \sqcap_\ell \mathbb{D}_\ell(\beta_2) \sqcap_\ell \dots \sqcap_\ell \mathbb{D}_\ell(\beta_n)$ ($\mathbb{D}_\ell(\beta)$ means the representation of β in \mathbb{D}_ℓ , when α_ℓ exists in the leaf abstract domain \mathbb{D}_ℓ , we can use $\alpha_\ell(\beta)$ instead).

```

join(t1, t2 : binary decision trees, bound =  $\top$ )
{
  if (t1 == (l1)) && t2 == (l2)) then return (t1  $\sqcup_\ell$  t2)  $\sqcap_\ell$  bound;

  let t1 = [[B: t1l, t1r]] and t2 = [[B: t2l, t2r]];
  return [[B: join(t1l, t2l, bound  $\sqcap_\ell$   $\mathbb{D}_\ell(B)$ ),
            join(t1r, t2r, bound  $\sqcap_\ell$   $\mathbb{D}_\ell(\neg B)$ )]];
}

```

Example 6.2.2. We have $\llbracket x \leq 50 : (x = 0 \wedge y = 0), (\perp_\ell) \rrbracket \sqsubseteq \llbracket x \leq 50 : (0 \leq x \leq 1 \wedge x = y), (\perp_\ell) \rrbracket$ and $\llbracket x \leq 50 : (x = 0 \wedge y = 0), (\perp_\ell) \rrbracket \not\sqsubseteq \llbracket x \leq 50 : (x = 1 \wedge y = 1), (\perp_\ell) \rrbracket$. \square

6.3 Transfer Functions

We define transfer functions for both tests and assignments. The tests either occur in a loop head or occur in the branch. Hence, we define both loop test transfer function and branch test transfer function for the binary decision tree abstract domain functor.

6.3.1 Loop Test Transfer Function

The transfer function for the loop tests is simple. Given a binary decision tree $t \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$ and a loop test B , we define $t \sqcap_t B$ as:

$$\begin{aligned}
 \perp_t \sqcap_t B &\triangleq \perp_t \\
 \top_t \sqcap_t B &\triangleq (B) \\
 t \sqcap_t \text{false} &\triangleq \perp_t \\
 t \sqcap_t \text{true} &\triangleq t \\
 (p) \sqcap_t B &\triangleq (p \sqcap_\ell \mathbb{D}_\ell(B)) \\
 \llbracket B' : t_l, t_r \rrbracket \sqcap_t B &\triangleq \llbracket B' : t_l \sqcap_t \mathbb{D}_\ell(B' \wedge B), t_r \sqcap_t \mathbb{D}_\ell(\neg B' \wedge B) \rrbracket
 \end{aligned}$$

Then the transfer function $f_L \llbracket B \rrbracket t$ for the loop test B of the binary decision tree t can be defined as:

$$f_L \llbracket B \rrbracket t \triangleq t \sqcap_t B.$$

Example 6.3.1. Let t be the binary decision tree in Example 6.1.2, then $f_L \llbracket y \geq 0 \rrbracket t = \llbracket x \leq 50 : (0 \leq x \leq 50 \wedge x = y), (51 \leq x \leq 102 \wedge x + y - 102 = 0) \rrbracket$. \square

6.3.2 Branch Test Transfer Function

The binary decision tree can be constructed in two different ways. On one hand, it can be generated immediately after the set \mathcal{B} of branch condition paths has been generated

in the pre-analysis. In this way, all leaves of the binary decision tree will be set to \top_ℓ for the first program point and \perp_ℓ for others ($\top_\ell, \perp_\ell \in \mathbb{D}_\ell$) at the beginning. On the other hand, both binary decision tree and \mathcal{B} can be constructed on the fly during the static analysis. In this last case, we have $\mathcal{B} = \emptyset$ and the binary decision tree $t = (\top_\ell)$ for the first program point and $t = (\perp_\ell)$ for others at the beginning.

In the latter case, the branch test transfer function should first construct the new binary decision tree from the old one by splitting on the branch condition when it has been first met in the analysis. Given a binary decision condition $t \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$ and a branch test B that's been first met, there are two situations.

One situation is that the branch condition B is independent, that is, it does not occur inside any scope of a branch. In this situation, the new binary decision tree t' can be constructed by replacing each leaf p in the binary tree t with a subtree $\llbracket B : (p \sqcap_\ell \mathbb{D}_\ell(B)), (p \sqcap_\ell \mathbb{D}_\ell(\neg B)) \rrbracket$. We also have $\mathcal{B}' = \{\underline{\pi}_b \cdot B \mid \underline{\pi}_b \in \mathcal{B}\} \cup \{\underline{\pi}_b \cdot \neg B \mid \underline{\pi}_b \in \mathcal{B}\}$.

The other situation is that the branch condition B is inside a scope of a branch. Let B' be the condition of the branch and there is no other branch scope between B and B' , if B is inside the true branch of B' , then the new binary decision tree t' can be constructed by replacing each left leaf p of B' in the binary tree t with a subtree $\llbracket B : (p \sqcap_\ell \mathbb{D}_\ell(B)), (p \sqcap_\ell \mathbb{D}_\ell(\neg B)) \rrbracket$. We also have $\mathcal{B}' = \{\underline{\pi}_b \cdot B' \cdot B \mid \underline{\pi}_b \cdot B' \in \mathcal{B}\} \cup \{\underline{\pi}_b \cdot B' \cdot \neg B \mid \underline{\pi}_b \cdot B' \in \mathcal{B}\} \cup (\mathcal{B} \setminus \{\underline{\pi}_b \cdot B' \mid \underline{\pi}_b \cdot B' \in \mathcal{B}\})$. If B is inside the false branch of B' , the right leaves of B' instead of left leaves should be replaced by the same subtrees and $\mathcal{B}' = \{\underline{\pi}_b \cdot \neg B' \cdot B \mid \underline{\pi}_b \cdot \neg B' \in \mathcal{B}\} \cup \{\underline{\pi}_b \cdot \neg B' \cdot \neg B \mid \underline{\pi}_b \cdot \neg B' \in \mathcal{B}\} \cup (\mathcal{B} \setminus \{\underline{\pi}_b \cdot \neg B' \mid \underline{\pi}_b \cdot \neg B' \in \mathcal{B}\})$.

Then in both ways, the branch test transfer function will do the same thing as loop test transfer function. Given the branch test B and the binary decision tree $t \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$, we have:

$$f_B \llbracket B \rrbracket t \triangleq t \sqcap_t B.$$

Example 6.3.2. Let t be the binary decision tree in Example 6.1.2, then $f_B \llbracket x \leq 50 \rrbracket t = \llbracket x \leq 50 : (0 \leq x \leq 50 \wedge x = y), (\perp_\ell) \rrbracket$. \square

6.3.3 Assignment Transfer Function

Given a binary decision tree $t \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$, the assignment $x = E$ can be performed at each leaf in t by using the assignment transfer function of \mathbb{D}_ℓ . E.g., let $t = \llbracket x \leq 50 : (0 \leq x \leq 50), (\perp_\ell) \rrbracket$ and given an assignment $x = x + 1$, after performing the assignment transfer function of Polyhedra abstract domain on each leaf of t , we will get $t' = \llbracket x \leq 50 : (1 \leq x \leq 51), (\perp_\ell) \rrbracket$.

Generally, the branch condition paths in \mathcal{B} are used as labels separating the abstract properties in disjunction which are gathered in the leaves. But this is not always the case. For example, in the join operator, we use the branch conditions in \mathcal{B} to reduce the result of the join. After performing the assignment transfer function of leaf abstract domain \mathbb{D}_ℓ on each leaf, we may also need to manipulate the leaves using the branch condition paths in \mathcal{B} .

Let's check the above result t' after the assignment, it appears that some leaves in the new binary decision tree may not satisfy some branch conditions in the branch condition paths which are leading to them. For example, $1 \leq x \leq 51$ is not satisfying the branch condition $x \leq 50$. We know the violation part is actually satisfying the negation of those branch conditions. Hence we need to use the branch condition $x \leq 50$ to separate $1 \leq x \leq 51$ into $1 \leq x \leq 50 \vee x = 51$ and update the corresponding leaves. For example, we have $t'' = \llbracket x \leq 50 : (1 \leq x \leq 50), (x = 51) \rrbracket$.

Let's see another example. Assume $t = \llbracket x \text{ is odd} : (x = [1, 1]), (x = [2, 2]) \rrbracket$ where the leaf abstract domain \mathbb{D}_ℓ is the interval abstract domain and the assignment is $x = x + 1$. After performing the assignment transfer function of the interval abstract domain on each

leaf of t , we will get $t' = \llbracket x \text{ is odd} : (\langle x = [2, 2] \rangle), (\langle x = [3, 3] \rangle) \rrbracket$. In this case, we will see that both abstract properties in the left leaf and in the right leaf are violating the branch condition “ x is odd”. Hence we need to exchange the abstract properties in the left leaf and in the right leaf which we will get $t'' = \llbracket x \text{ is odd} : (\langle x = [3, 3] \rangle), (\langle x = [2, 2] \rangle) \rrbracket$.

We call these kind of procedures *reconstruction on leaves*. Given a binary decision tree t after an assignment, we define the procedure of reconstruction on leaves as follow:

1. Collecting all leave properties in t , let it be $\{p_1, p_2, \dots, p_n\}$;
2. For each leaf in t , let $\pi_b = \beta_1 \cdot \beta_2 \cdot \dots \cdot \beta_n$ be the branch condition path leading to it.

We then calculate $p'_i = p_i \sqcap_\ell (\mathbb{D}_\ell(\beta_1 \wedge \beta_2 \wedge \dots \wedge \beta_n))$.

3. For each leaf in t , update it with $p'_1 \sqcup_\ell p'_2 \sqcup_\ell \dots \sqcup_\ell p'_n$.

Correctness. Let $p = p_1 \vee p_2 \vee \dots \vee p_n$ be the disjunction of all properties in leaves before reconstruction on leaves. For each leaf ℓ_i in t , we have $\ell_i = (p_1 \sqcap_\ell (\mathbb{D}_\ell(\beta_1^i \wedge \beta_2^i \wedge \dots \wedge \beta_n^i))) \sqcup_\ell \dots \sqcup_\ell (p_n \sqcap_\ell (\mathbb{D}_\ell(\beta_1^i \wedge \beta_2^i \wedge \dots \wedge \beta_n^i))) = (p_1 \sqcup_\ell \dots \sqcup_\ell p_n) \sqcap_\ell (\mathbb{D}_\ell(\beta_1^i \wedge \beta_2^i \wedge \dots \wedge \beta_n^i))$ after reconstruction on leaves. We then have the disjunction of all properties in leaves after reconstruction on leaves is $p' = \ell_1 \vee \dots \vee \ell_n = (p_1 \sqcup_\ell \dots \sqcup_\ell p_n) \sqcap_\ell (\mathbb{D}_\ell(\beta_1^1 \wedge \beta_2^1 \wedge \dots \wedge \beta_n^1)) \vee \dots \vee (p_1 \sqcup_\ell \dots \sqcup_\ell p_n) \sqcap_\ell (\mathbb{D}_\ell(\beta_1^n \wedge \beta_2^n \wedge \dots \wedge \beta_n^n)) = (p_1 \sqcup_\ell \dots \sqcup_\ell p_n) \sqcap_\ell ((\mathbb{D}_\ell(\beta_1^1 \wedge \beta_2^1 \wedge \dots \wedge \beta_n^1)) \vee \dots \vee (\mathbb{D}_\ell(\beta_1^n \wedge \beta_2^n \wedge \dots \wedge \beta_n^n))) = (p_1 \sqcup_\ell \dots \sqcup_\ell p_n) \sqcap_\ell \text{true} = p_1 \sqcup_\ell \dots \sqcup_\ell p_n \equiv p$. This shows that the reconstruction on leaves procedure will not change the result of the assignment transfer function.

6.4 Extrapolation Operators

When the leaf abstract domain \mathbb{D}_ℓ has strictly increasing and/or strictly decreasing infinite chains, widening and/or narrowing operators are required in the binary decision tree abstract domain functor to accelerate the convergence of fixpoint iterates.

6.4.1 Widening

Given two binary decision trees $t_1, t_2 \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$, the widening $t = t_1 \nabla_t t_2$ can be computed using the widening ∇_ℓ in the leaf abstract domain \mathbb{D}_ℓ similar to the join operator, that is, computing the widening $\ell_1 \nabla_\ell \ell_2$ for each pair (ℓ_1, ℓ_2) of leaves in (t_1, t_2) where ℓ_1 and ℓ_2 are led by the same branch condition path $\underline{\pi}_b \in \mathcal{B}$ while the branch conditions in $\underline{\pi}_b$ are also used as the threshold. Let $\underline{\pi}_b = \beta_1 \cdot \beta_2 \cdot \dots \cdot \beta_n$ where $\beta_i = B_i$ or $\neg B_i, i = 1, \dots, n$, we have each leaf $\ell = (\ell_1 \nabla_\ell \ell_2) \sqcap_\ell \mathbb{D}_\ell(\beta_1) \sqcap_\ell \mathbb{D}_\ell(\beta_2) \sqcap_\ell \dots \sqcap_\ell \mathbb{D}_\ell(\beta_n)$.

```
widening(t1, t2 : binary decision trees, bound =  $\top$ )
{
  if (t1 == (l1) && t2 == (l2)) then return (t1  $\nabla_\ell$  t2)  $\sqcap_\ell$  bound;

  let t1 = [[B: t1l, t1r]] and t2 = [[B: t2l, t2r]];
  return [[B: widening(t1l, t2l, bound  $\sqcap_\ell$   $\mathbb{D}_\ell(B)$ ),
          widening(t1r, t2r, bound  $\sqcap_\ell$   $\mathbb{D}_\ell(\neg B)$ )]];
}
```

6.4.2 Narrowing

The narrowing operator in the binary decision tree abstract domain functor is very similar to its meet operator. Given two binary decision trees $t_1, t_2 \in \mathbb{T}(\mathcal{B}, \mathbb{D}_\ell)$, the narrowing $t = t_1 \Delta_t t_2$ can be computed using the narrowing Δ_ℓ in the leaf abstract domain \mathbb{D}_ℓ . Let ℓ_1, ℓ_2 are leaves of t_1, t_2 respectively, where the same branch condition path $\underline{\pi}_b \in \mathcal{B}$ leads to ℓ_1 and ℓ_2 , then $\ell = \ell_1 \Delta_\ell \ell_2$ is the leaf of t led by the same branch condition path $\underline{\pi}_b \in \mathcal{B}$. After computing each leaf $\ell = \ell_1 \Delta_\ell \ell_2$ in t , we then get $t = t_1 \Delta_t t_2$.


```

narrowing(t1, t2 : binary decision trees)
{
    if (t1 == (l1)) && t2 == (l2)) then return t1 Δℓ t2;

    let t1 = [[B: t1l, t1r]] and t2 = [[B: t2l, t2r]];
    return [[B: narrowing(t1l, t2l), narrowing(t1r, t2r)]];
}

```

Example 6.4.1. Let $t_1 = \llbracket x \leq 50 : (x = 0 \wedge y = 0), (\perp_\ell) \rrbracket$ and $t_2 = \llbracket x \leq 50 : (x = y \wedge 0 \leq x \leq 1), (\perp_\ell) \rrbracket$. It's easy to see that $t_1 \subseteq t_2$. In polyhedra, we have $(x = 0 \wedge y = 0) \nabla_t (x = y \wedge 0 \leq x \leq 1) = x \geq 0 \wedge x = y$. Hence, we have $t_1 \nabla_t t_2 = \llbracket x \leq 50 : (0 \leq x \leq 50 \wedge x = y), (\perp_\ell) \rrbracket$. \square

Note that we assumed the two binary decision trees in the meet, join, widening and narrowing would have the same shape. This is always true if we generate the set of branch condition paths \mathcal{B} in the pre-analysis. In this case, the binary decision trees generated from \mathcal{B} will always have the same shape. If we generate the set of branch condition paths \mathcal{B} on the fly in the analysis, the two binary decision trees in the meet, join, widening and narrowing may not always have the same shape. When the two binary decision trees have different shape, it's always the case that one binary decision tree is smaller than the other one and each path of it from root to leaf is the prefix of some paths from root to leaves of the other one's. Hence it can be easily split to match the shape of the other one.

6.5 Other Operators

Although the number of branch conditions in a program is always finite, it may still be a very large number. A large number of branch conditions means a large binary decision tree, with a potentially an exponential growth which is not acceptable in practice. Hence, we need limit the size (depth) of the binary decision trees.

One method is to eliminate decision nodes by merging their subtrees when the binary decision tree grows too deep. This can be done as follow:

1. Pick up a branch condition B . We can simply use the one at the root, or the nearest one to the leaves, or at random. We can also design a ranking function based on the information from the analysis for each branch condition to estimate how likely it is to be eliminated with minimal information loss. Then we always choose the most likely one.
2. Eliminate B (B or $\neg B$) from each branch condition path in \mathcal{B} .
3. For each subtree of the form $\llbracket B : t_t, t_f \rrbracket$, if t_t and t_f have identical decision nodes, replace it by $t_t \sqcup_t t_f$.
4. Otherwise, there are decision nodes existing only in t_t or t_f . For each of those decision nodes, (recursively) eliminate it by merging its subtrees. When no such decision node exists, we get t'_t and t'_f , and they must have identical decision nodes, so $\llbracket B : t_t, t_f \rrbracket$ can be replaced by $t'_t \sqcup_t t'_f$.

Another method is to generate a smaller \mathcal{B} by abstracting the branch condition paths in \mathcal{B} into shorter ones. We may partition the set of branch conditions by its appearance inside or outside loops and then only keep the ones appear inside the loops in \mathcal{B} . We may also only keep the branch conditions which have some particular form, such as $ax \leq b$, etc.

The second method is different from the first one because it can be done in the pre-analysis or on the fly before splitting trees, thus no merging is needed during the analysis. This reduces the cost of the analysis, thus improves its efficiency. But because all the branch conditions being eliminated are not based on the information that is collected during the static analysis, the result may be less precise than the one generated from the first method. Moreover, eliminating branch conditions and merging their subtrees allow us to dynamically change the binary decision trees on the fly. This provides a more flexible way of adjusting the cost/precision ratio of the static analysis.

6.6 Example

Let us come back to Example 6.1.1. We choose the polyhedra abstract domain as the leaf abstract domain and we have $\mathcal{B} = \{x \leq 50, \neg(x \leq 50)\}$. Initially, we set $t = (\perp_\ell)$ in the program point ¹. After the assignment “ $x = 0; y = 0;$ ”, we have “ $t = (x = 0 \wedge y = 0)$ ”. Let t_i be the abstract property at program point ¹ after the i -th iteration, then $t_0 = (x = 0 \wedge y = 0)$. In first iteration, we have to construct the binary decision tree when first reaching the branch test “ $x \leq 50$ ”. In this case, we have $t'_0 = \llbracket x \leq 50 : (x = 0 \wedge y = 0), (\perp_\ell) \rrbracket$. At the end of the first iteration, we get $t''_0 = \llbracket x \leq 50 : (x = 1 \wedge y = 1), (\perp_\ell) \rrbracket$. Then $t_1 = t_0 \cup_t t''_0 = \llbracket x \leq 50 : (x = y \wedge 0 \leq x \leq 1), (\perp_\ell) \rrbracket$. Afterward, we apply the widening and get $t'_1 = t_0 \nabla t_1 = \llbracket x \leq 50 : (0 \leq x \leq 50 \wedge x = y), (\perp_\ell) \rrbracket$. In the second iteration, the assignment “ $x++;$ ” leads to reconstruction on leaves, hence we get $t''_1 = \llbracket x \leq 50 : (1 \leq x \leq 50 \wedge x = y), (x = 51 \wedge y = 51) \rrbracket$. Then $t_2 = t_1 \cup_t t''_1 = \llbracket x \leq 50 : (0 \leq x \leq 50 \wedge x = y), (x = 51 \wedge y = 51) \rrbracket$. After the third iteration, $t_3 = \llbracket x \leq 50 : (0 \leq x \leq 50 \wedge x = y), (x + y - 102 = 0 \wedge 51 \leq x \leq 52) \rrbracket$. We then apply the widening and get $t'_3 = t_2 \nabla t_3 = \llbracket x \leq 50 : (0 \leq x \leq 50 \wedge x = y), (x + y - 102 = 0 \wedge x \geq 51) \rrbracket$. One more iteration yields $t_4 = \llbracket x \leq 50 : (0 \leq x \leq 50 \wedge x = y), (x + y - 102 = 0 \wedge 51 \leq x \leq 103) \rrbracket$.

It follows that the program analysis converges. Hence t_4 is the invariant at program point 1.

6.7 Related Work

A systematic characterization of the least bases for the disjunctive completion of abstract domains can be found in [GR98]. The trace partitioning using control flows was first introduced in [Cou81]. A static analysis framework via trace partitioning was proposed by [HT98]. In this framework, the control flow is used to choose which disjunctions to keep but it lacks the merge of partitions, which may lead to exponential cost. In [MR05], a trace partitioning domain, where the partitioning of traces are based on the history of the control flow, has been proposed. The main difference between their partitioning and ours is we made the partitioning by the syntactic analysis from the control flow of the program while they made the partitioning manually which is based on the input.

Decision trees have been used for the disjunctive refinement of an abstract domain such as [GC10] for the interval abstract domain based on decision trees. A general segmented decision tree abstract domain, where disjunctions are determined by values of variables is introduced in [CCM10b]. Moreover, [UM14] proposed a general disjunctive refinement of an abstract domain based on decision trees extended with linear constraints for program termination. The difference between those works and ours is their partitionings are mainly based on the value of some variables while ours are directly based on the branch conditions.

There also exist several works on directly allowing disjunction in the domain, i.e., powerset domain [BHZ07]. In [SISG06], the disjunctions are computed on an elaboration, which can be viewed as a multiply duplication, of the programs CFG structure. Moreover, our binary decision tree abstract domain functor can also be useful to scale traditional path-sensitive program analysis [WZH⁺13].

Conclusion

The static analysis by abstract interpretations is very efficient by using convex abstract domains, but lacks of expressiveness. On the other hand, the first-order logic and related SMT solvers and other theorem provers are very good at expressiveness. Hence, it will be interesting to exploit for logical abstract interpretations and domains. In this thesis, two logical abstract domains were presented to address this issue. In chapter 3, we investigated finite conjunctions of affine equalities and a set of logical operations to manipulate them. SMT solvers are used for the computation of there logical operations as well as transformations which are necessary to define an abstract domain. In chapter 4, more complicate first-order formulas - finite conjunctions of linear inequalities - have been investigated which include how to use SMT solvers to compute transformations and other necessary logical operations as well.

Abstract domains are often using convex sets which are conjunctions of linear constraints to represent program properties. This convexity makes the static analysis efficient and scalable. On the other hand, it may cause rough approximations and produce much less precise results. In practice, it is often necessary to refine the abstract domains by allowing weak forms of disjunctions to increasing the precision. In this thesis, we also presented an abstract domain functor to contribute on this issue. Our abstract domain functor uses binary decision trees to represent and manipulate program invariants. It is parameterized by decision nodes which are a set of boolean tests appearing in the pro-

grams and by a numerical abstract domain whose elements are the leaves. In chapter 5, we introduced the branch condition path abstraction which defines a kind of trace partitioning on the concrete level (trace semantics of program). It decides the decision nodes in the binary decision trees. In chapter 6, we defined our binary decision tree abstract domain functor and discussed the implementation of it by providing algorithms for transformations and other useful domain operations. This binary decision tree abstract domain functor can provide a flexible way of adjusting the cost/precision ratio for static analysis.

Future Work

In this thesis, we only investigated the use of SMT solvers in logical abstract interpretations and domains. It is also interesting to exploit abstract interpretation in logical / theory approach using SMT solvers and theorem provers. Moreover, we want to investigate the combination of algebraic and logical abstract interpretations which may provide scalability, expressivity, natural interface with the end-user using logical formulas, and soundness with respect to the program semantics.

Our binary decision tree abstract domain functor relies on trace partitioning for adjusting the cost/precision ratio. Hence, we'd like to investigate new methods for partitioning traces in different levels. In this thesis, we only consider numerical abstract domains in the binary decision tree abstract domain functor. We also want to include the symbolic abstract domains, and even the logical abstract domains we defined in this thesis. Last, it is worth to investigate new methods that allow more disjunctions in abstract domains while minimize the cost of the analysis.

Appendix A

Proof of Theorem 1.2.3

The proof is by structural induction on the syntax of the command C.

– skip command

$$\begin{aligned} & \alpha^a(\mathcal{S}^t[\text{skip}]) \\ = & \alpha^a(\{\langle \text{skip}, \rho \rangle \xrightarrow{\text{skip}} \langle \text{skip}, \rho \rangle \mid \rho \in \mathcal{E}\}) && \{\text{definition of } \mathcal{S}^t[\text{skip}]\} \\ = & \{\text{skip}\} && \{\text{definition of } \alpha^a\} \\ \triangleq & \mathcal{G}^a[\circ \rightarrow \boxed{\text{skip}} \rightarrow \circ] && \{\text{definition of } \mathcal{G}^a[\circ \rightarrow \boxed{\text{skip}} \rightarrow \circ]\} \end{aligned}$$

– Assignment

$$\begin{aligned} & \alpha^a(\mathcal{S}^t[x = E]) \\ = & \alpha^a(\{\langle \text{skip}, \rho \rangle \xrightarrow{x=E} \langle \text{stop}, \rho[x := v] \rangle \mid \rho \in \mathcal{E} \wedge v \in \mathcal{E}[E]\rho\}) && \{\text{definition of } \mathcal{S}^t[x = E]\} \\ = & \{x = E\} && \{\text{definition of } \alpha^a\} \\ \triangleq & \mathcal{G}^a[\circ \rightarrow \boxed{x := E} \rightarrow \circ] && \{\text{definition of } \mathcal{G}^a[\circ \rightarrow \boxed{x := E} \rightarrow \circ]\} \end{aligned}$$

– **Conditional**

$$\begin{aligned}
& \alpha^a(\mathcal{S}^t[\text{if (B) } \{C_1\} \text{ else } \{C_2\}]) \\
&= \alpha^a(\{\langle \text{if (B) } \{C_1\} \text{ else } \{C_2\}, \rho \rangle \xrightarrow{B} \langle C_1, \rho \rangle \xrightarrow{A} \pi \mid \rho \in \mathcal{E} \wedge \text{true} \in \mathcal{E}[\text{B}]\rho \wedge \langle C_1, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^t[\text{C}_1]\} \cup \{\langle \text{if (B) } \{C_1\} \text{ else } \{C_2\}, \rho \rangle \xrightarrow{\neg B} \langle C_2, \rho \rangle \xrightarrow{A} \pi \mid \rho \in \mathcal{E} \wedge \text{false} \in \mathcal{E}[\text{B}]\rho \wedge \langle C_2, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^t[\text{C}_2]\}) \\
& \quad \text{\textit{\textless definition of } \mathcal{S}^t[\text{if (B) } \{C_1\} \text{ else } \{C_2\}]\textit{\textless}} \\
&= \alpha^a(\{\langle \text{if (B) } \{C_1\} \text{ else } \{C_2\}, \rho \rangle \xrightarrow{B} \langle C_1, \rho \rangle \xrightarrow{A} \pi \mid \rho \in \mathcal{E} \wedge \text{true} \in \mathcal{E}[\text{B}]\rho \wedge \langle C_1, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^t[\text{C}_1]\}) \cup \alpha^a(\{\langle \text{if (B) } \{C_1\} \text{ else } \{C_2\}, \rho \rangle \xrightarrow{\neg B} \langle C_2, \rho \rangle \xrightarrow{A} \pi \mid \rho \in \mathcal{E} \wedge \text{false} \in \mathcal{E}[\text{B}]\rho \wedge \langle C_2, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^t[\text{C}_2]\}) \\
& \quad \text{\textit{\textless since } \alpha^a(S \cup S') = \alpha^a(S) \cup \alpha^a(S')\textit{\textless}} \\
&\subseteq \{B \cdot \alpha^a(\langle C_1, \rho \rangle \xrightarrow{A} \pi \mid \langle C_1, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^t[\text{C}_1])\} \cup \{\neg B \cdot \alpha^a(\langle C_2, \rho \rangle \xrightarrow{A} \pi \mid \langle C_2, \rho \rangle \xrightarrow{A} \pi \in \mathcal{S}^t[\text{C}_2])\}) \\
& \quad \text{\textit{\textless definition of } \alpha^a \textit{\textless and ignoring the test}\textit{\textless}} \\
&= \{B\} \cdot \{\alpha^a(\pi') \mid \pi' \in \mathcal{S}^t[\text{C}_1]\}) \cup \{\neg B\} \cdot \{\alpha^a(\pi'') \mid \pi'' \in \mathcal{S}^t[\text{C}_2]\})
\end{aligned}$$

\textit{\textless by defining the set of action paths concatenation } S \cdot S' \triangleq \{\varpi \cdot \varpi' \mid \varpi \in S \wedge \varpi' \in S'\textit{\textless, letting } \pi' = \langle C_1, \rho \rangle \xrightarrow{A} \pi\textit{\textless, and } \pi'' = \langle C_2, \rho \rangle \xrightarrow{A} \pi\textit{\textless}} \\
= \{B\} \cdot \alpha^a(\mathcal{S}^t[\text{C}_1]) \cup \{\neg B\} \cdot \alpha^a(\mathcal{S}^t[\text{C}_2]) \quad \text{\textit{\textless definition of } \alpha^a\textit{\textless}} \\
\subseteq \{B\} \cdot \mathcal{G}^a[\text{O} \rightarrow \boxed{\text{C}_1} \rightarrow \text{O}] \cup \{\neg B\} \cdot \mathcal{G}^a[\text{O} \rightarrow \boxed{\text{C}_2} \rightarrow \text{O}]

\textit{\textless by structural induction hypothesis and the set of action paths concatenation } \cdot \textit{\textless is \textit{\textless-increasing in both of its arguments.\textit{\textless}}

$$\triangleq \mathcal{G}^a[\text{O} \rightarrow \boxed{\text{B}} \begin{array}{l} \xrightarrow{\text{tt}} \boxed{\text{C}_1} \\ \xrightarrow{\text{ff}} \boxed{\text{C}_2} \end{array} \rightarrow \text{O}] \quad \text{\textit{\textless definition of } \mathcal{G}^a[\text{O} \rightarrow \boxed{\text{B}} \begin{array}{l} \xrightarrow{\text{tt}} \boxed{\text{C}_1} \\ \xrightarrow{\text{ff}} \boxed{\text{C}_2} \end{array} \rightarrow \text{O}]\textit{\textless}}$$

– **Sequence, finite executions**

$$\begin{aligned}
& \alpha^a(\mathcal{S}^{t*}[\text{C}_1; \text{C}_2]) \\
&= \alpha^a(\{(\pi \text{ ; } C_2) \xrightarrow{A} \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \mid \pi \xrightarrow{A} \langle \text{stop}, \rho \rangle \in \mathcal{S}^{t*}[\text{C}_1] \wedge \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \in \mathcal{S}^{t*}[\text{C}_2]\}) \\
& \quad \text{\textit{\textless definition of } \mathcal{S}^{t*}[\text{C}_1; \text{C}_2]\textit{\textless}}
\end{aligned}$$

$$\begin{aligned}
&= \alpha^a(\{\pi \xrightarrow{A} \langle \text{stop}, \rho \rangle \xrightarrow{A'} \pi' \mid \pi \xrightarrow{A} \langle \text{stop}, \rho \rangle \in \mathcal{S}^{t*}[[C_1]] \wedge \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \in \mathcal{S}^{t*}[[C_2]]\}) \\
&\qquad\qquad\qquad \wr \text{since } \alpha^a(\pi \ ; \ C) = \alpha^a(\pi) \wr \\
&= \{\alpha^a(\pi) \cdot A \cdot A' \cdot \alpha^a(\pi') \mid \pi \xrightarrow{A} \langle \text{stop}, \rho \rangle \in \mathcal{S}^{t*}[[C_1]] \wedge \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \in \mathcal{S}^{t*}[[C_2]]\} \\
&\qquad\qquad\qquad \wr \text{since } \alpha^a(\pi \xrightarrow{A} \pi') = \alpha^a(\pi) \cdot A \cdot \alpha^a(\pi') \wr \\
&= \{\alpha^a(\pi'') \cdot \alpha^a(\pi''') \mid \alpha^a(\pi'') \in \mathcal{S}^{t*}[[C_1]] \wedge \alpha^a(\pi''') \in \mathcal{S}^{t*}[[C_2]]\} \\
&\qquad\qquad\qquad \wr \text{letting } \alpha^a(\pi'') = \pi \xrightarrow{A} \langle \text{stop}, \rho \rangle \text{ and } \alpha^a(\pi''') = \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \wr \\
&= \alpha^a(\mathcal{S}^{t*}[[C_1]]) \cdot \alpha^a(\mathcal{S}^{t*}[[C_2]]) \quad \wr \text{definition of } \alpha^a \text{ and set of action paths concatenation} \wr \\
&\subseteq \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_1} \rightarrow \circ] \cdot \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_2} \rightarrow \circ] \\
&\qquad\qquad\qquad \wr \text{by structural induction hypothesis and the set of action paths concatenation } \cdot \text{ is} \\
&\qquad\qquad\qquad \subseteq\text{-increasing in both of its arguments.} \wr \\
&\triangleq \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_1} \rightarrow \boxed{C_2} \rightarrow \circ] \quad \wr \text{definition of } \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_1} \rightarrow \boxed{C_2} \rightarrow \circ] \wr
\end{aligned}$$

– **Sequence, infinite executions**

$$\begin{aligned}
&\alpha^a(\mathcal{S}^{t\infty}[[C_1 ; C_2]]) \\
&= \alpha^a(\{(\pi \ ; \ C_2) \mid \pi \in \mathcal{S}^{t\infty}[[C_1]]\} \cup \{(\pi \ ; \ C_2) \xrightarrow{A} \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \mid \rho \in \mathcal{E} \wedge \pi \xrightarrow{A} \langle \text{stop}, \rho \rangle \in \\
&\quad \mathcal{S}^{t*}[[C_1]] \wedge \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \in \mathcal{S}^{t\infty}[[C_2]]\}) \quad \wr \text{definition of } \mathcal{S}^{t\infty}[[C_1 ; C_2]] \wr \\
&= \alpha^a(\{(\pi \ ; \ C_2) \mid \pi \in \mathcal{S}^{t\infty}[[C_1]]\}) \cup \alpha^a(\{(\pi \ ; \ C_2) \xrightarrow{A} \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \mid \rho \in \mathcal{E} \wedge \pi \xrightarrow{A} \langle \text{stop}, \rho \rangle \in \\
&\quad \mathcal{S}^{t*}[[C_1]] \wedge \langle C_2, \rho \rangle \xrightarrow{A'} \pi' \in \mathcal{S}^{t\infty}[[C_2]]\}) \quad \wr \text{since } \alpha^a \text{ preserves unions} \wr \\
&= \alpha^a(\mathcal{S}^{t\infty}[[C_1]]) \cup \alpha^a(\mathcal{S}^{t*}[[C_1]]) \cdot \alpha^a(\mathcal{S}^{t\infty}[[C_2]]) \\
&\qquad\qquad\qquad \wr \text{since } \alpha^a(\pi \ ; \ C_2) = \alpha^a(\pi) \text{ and } \alpha^a(\pi \xrightarrow{A} \pi') = \alpha^a(\pi) \cdot A \cdot \alpha^a(\pi') \wr \\
&\subseteq \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C_1} \rightarrow \circ] \cup \mathcal{G}^{a*}[\circ \rightarrow \boxed{C_1} \rightarrow \circ] \cdot \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C_2} \rightarrow \circ]
\end{aligned}$$

by structural induction hypothesis and the set of action paths concatenation \cdot is \subseteq -increasing in both of its arguments.

$$\triangleq \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C_1} \rightarrow \boxed{C_2} \rightarrow \circ] \quad \wr \text{definition of } \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C_1} \rightarrow \boxed{C_2} \rightarrow \circ] \wr$$

– Iteration, transformer

$$\begin{aligned} & \alpha^a(\mathcal{F}^{ti}[\text{while (B) } \{C\}](X)) \\ &= \alpha^a(\{\langle \text{while (B) } \{C\}, \rho \rangle \mid \rho \in \mathcal{E}\} \cup \{\pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \xrightarrow{B} (\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \ ; \ \text{while (B) } \{C\} \mid \pi, \pi' \in \Pi^* \wedge \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \in X \wedge \text{true} \in \mathcal{E}[\boxed{B}]\rho \wedge (\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \in \mathcal{S}^{t*}[\boxed{C}]\}) \quad \wr \text{definition of } \mathcal{F}^{ti}[\text{while (B) } \{C\}] \wr \\ &= \alpha^a(\{\langle \text{while (B) } \{C\}, \rho \rangle \mid \rho \in \mathcal{E}\}) \cup \alpha^a(\{\pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \xrightarrow{B} (\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \ ; \ \text{while (B) } \{C\} \mid \pi, \pi' \in \Pi^* \wedge \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \in X \wedge \text{true} \in \mathcal{E}[\boxed{B}]\rho \wedge (\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \in \mathcal{S}^{t*}[\boxed{C}]\}) \quad \wr \text{since } \alpha^a(S \cup S') = \alpha^a(S) \cup \alpha^a(S') \wr \\ &\subseteq \{\varepsilon\} \cup \{\alpha^a(\pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \xrightarrow{B} (\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \ ; \ \text{while (B) } \{C\}) \mid \pi, \pi' \in \Pi^* \wedge \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \in X \wedge (\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \in \mathcal{S}^{t*}[\boxed{C}]\} \\ & \quad \wr \text{definition of } \alpha^a \text{ and ignoring the result } \text{true} \in \mathcal{E}[\boxed{B}]\rho \text{ of tests} \wr \\ &= \{\varepsilon\} \cup \{\alpha^a(\pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle) \cdot B \cdot \alpha^a(\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \mid \pi, \pi' \in \Pi^* \wedge \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \in X \wedge (\langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle) \in \mathcal{S}^{t*}[\boxed{C}]\} \\ & \quad \wr \text{since } \alpha^a(\pi \xrightarrow{A} \pi') = \alpha^a(\pi) \cdot A \cdot \alpha^a(\pi') \text{ and } \alpha^a(\pi \ ; \ C) = \alpha^a(\pi) \wr \\ &= \{\varepsilon\} \cup \{\alpha^a(\pi'') \cdot B \cdot \alpha^a(\pi''') \mid \pi'' \in X \wedge \pi''' \in \mathcal{S}^{t*}[\boxed{C}]\} \\ & \quad \wr \text{letting } \pi'' = \pi \xrightarrow{A} \langle \text{while (B) } \{C\}, \rho \rangle \text{ and } \pi''' = \langle C, \rho \rangle \xrightarrow{A'} \pi' \xrightarrow{A''} \langle \text{stop}, \rho' \rangle \wr \\ &= \{\varepsilon\} \cup \alpha^a(X) \cdot \{B\} \cdot \alpha^a(\mathcal{S}^{t*}[\boxed{C}]) \\ & \quad \wr \text{definition of } \alpha^a \text{ and set of action paths concatenation} \wr \\ &\subseteq \{\varepsilon\} \cup \alpha^a(X) \cdot \{B\} \cdot \mathcal{G}^{a*}[\circ \rightarrow \boxed{C} \rightarrow \circ] \end{aligned}$$

by structural induction hypothesis and the set of action paths concatenation \cdot is

$$\begin{aligned} & \subseteq\text{-increasing in both of its arguments.} \} \\ = & \mathcal{F}^{ai} \llbracket \circ \rightarrow \boxed{\text{B}} \xrightarrow{\text{tt}} \boxed{\text{C}} \rightarrow \circ \rrbracket (\alpha^a(X)) \quad \} \text{definition of } \mathcal{F}^{ai} \llbracket \circ \rightarrow \boxed{\text{B}} \xrightarrow{\text{tt}} \boxed{\text{C}} \rightarrow \circ \rrbracket \} \end{aligned}$$

We have proved the semi-commutation property:

$$\forall X \in \wp(\Pi^*) : \tag{A.1}$$

$$\alpha^a(\mathcal{F}^{ti} \llbracket \text{while}(\text{B}) \{ \text{C} \} \rrbracket (X)) \subseteq \mathcal{F}^{ai} \llbracket \circ \rightarrow \boxed{\text{B}} \xrightarrow{\text{tt}} \boxed{\text{C}} \rightarrow \circ \rrbracket (\alpha^a(X))$$

– Union preservation

Let us show that the iteration transformer preserves unions.

$$\begin{aligned} & \mathcal{F}^{ai} \llbracket \circ \rightarrow \boxed{\text{B}} \xrightarrow{\text{tt}} \boxed{\text{C}} \rightarrow \circ \rrbracket \left(\bigcup_{i \in \Delta} S_i \right) \\ = & \{ \varepsilon \} \cup \left(\bigcup_{i \in \Delta} S_i \right) \cdot \{ \text{B} \} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{\text{C}} \rightarrow \circ \rrbracket \quad \} \text{definition of } \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{\text{C}} \rightarrow \circ \rrbracket \} \\ = & \{ \varepsilon \} \cup \bigcup_{i \in \Delta} (S_i \cdot \{ \text{B} \} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{\text{C}} \rightarrow \circ \rrbracket) \\ & \} \text{(since the concatenation } \cdot \text{ of sets of action paths preserves unions)} \\ = & \bigcup_{i \in \Delta} (\{ \varepsilon \} \cup S_i \cdot \{ \text{B} \} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{\text{C}} \rightarrow \circ \rrbracket) \quad \} \text{definition of } \cup \} \\ = & \bigcup_{i \in \Delta} \mathcal{F}^{ai} \llbracket \circ \rightarrow \boxed{\text{B}} \xrightarrow{\text{tt}} \boxed{\text{C}} \rightarrow \circ \rrbracket (S_i) \quad \} \text{definition of } \mathcal{F}^{ai} \llbracket \circ \rightarrow \boxed{\text{B}} \xrightarrow{\text{tt}} \boxed{\text{C}} \rightarrow \circ \rrbracket \} \end{aligned}$$

It follows that is \subseteq -increasing.

– Fixpoint approximation

A consequence of the semi-commutation property is that the abstraction of the concrete iterates $\langle S_t^n, n \in \mathbb{N} \rangle$ of $\mathcal{F}^{ti} \llbracket \text{while}(\text{B}) \{ \text{C} \} \rrbracket$ from \emptyset by α^a is over-approximated by the iterates $\langle S_a^n, n \in \mathbb{N} \rangle$ of $\mathcal{F}^{ai} \llbracket \circ \rightarrow \boxed{\text{B}} \xrightarrow{\text{tt}} \boxed{\text{C}} \rightarrow \circ \rrbracket$ in the abstract: $\forall n \in \mathbb{N} : \alpha^a(S_t^n) \subseteq S_a^n$. The proof is

by recurrence on $n \in \mathbb{N}$.

$$\begin{aligned}
& - \alpha^a(S_t^0) = \alpha^a(\emptyset) = \emptyset = S_a^0 \\
& - \alpha^a(S_t^{n+1}) = \alpha^a(\mathcal{F}^{ti}[\text{while (B) \{C\}}](S_t^n)) \quad \{\text{definition of iteration}\} \\
& \subseteq \mathcal{F}^{ai}[\text{loop}(\text{B} \xrightarrow{\text{tt}} \text{C})](\alpha^a(S_t^n)) \quad \{\text{semi-commutation property}\} \\
& \subseteq \mathcal{F}^{ai}[\text{loop}(\text{B} \xrightarrow{\text{tt}} \text{C})](S_t^n) \\
& \quad \{\text{by induction hypothesis } \alpha^a(S_t^n) \subseteq S_a^n \text{ and } \mathcal{F}^{ai}[\text{loop}(\text{B} \xrightarrow{\text{tt}} \text{C})] \text{ is } \subseteq\text{-increasing}\} \\
& = S_a^{n+1} \quad \{\text{definition of iteration}\}
\end{aligned}$$

It follows that we have the following fixpoint approximation,

$$\begin{aligned}
& \alpha^a(\text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while (B) \{C\}}]) \\
& = \alpha^a\left(\bigcup_{n \in \mathbb{N}} S_t^n\right) \quad \{\text{since } \mathcal{F}^{ti}[\text{while (B) \{C\}}] \text{ preserves unions}\} \\
& = \bigcup_{n \in \mathbb{N}} \alpha^a(S_t^n) \quad \{\text{since } \alpha^a \text{ preserves unions}\} \\
& \subseteq \bigcup_{n \in \mathbb{N}} S_a^n \quad \{\text{since } \forall n \in \mathbb{N} : \alpha^a(S_t^n) \subseteq S_a^n \text{ and definition of } \cup\} \\
& = \text{lfp}^{\subseteq} \mathcal{F}^{ai}[\text{loop}(\text{B} \xrightarrow{\text{tt}} \text{C})] \quad \{\text{since } \mathcal{F}^{ai}[\text{loop}(\text{B} \xrightarrow{\text{tt}} \text{C})] \text{ preserves unions}\}
\end{aligned}$$

– Iterates and least fixpoint of the iteration transformer

Given a set \mathcal{A} of action paths, let us define its natural powers as $\mathcal{A}^0 \triangleq \{\varepsilon\}$, $\mathcal{A}^1 \triangleq \mathcal{A}$, $\mathcal{A}^n = \underbrace{\mathcal{A} \cdot \dots \cdot \mathcal{A}}_{n \text{ times}}$ for $n > 1$. The rule of powers $\mathcal{A}^{p+q} = \mathcal{A}^p \cdot \mathcal{A}^q$ is trivial.

Let us calculate the iterates $\langle F^n, n \in \mathbb{N} \rangle$ of $\mathcal{F}^{ai}[\text{loop}(\text{B} \xrightarrow{\text{tt}} \text{C})]$ from $F^0 = \emptyset$.

$$\begin{aligned}
& - F^1 = \mathcal{F}^{ai}[\text{loop}(\text{B} \xrightarrow{\text{tt}} \text{C})](F^0) = \{\varepsilon\} \\
& \quad \{\text{definition of iteration, } \mathcal{F}^{ai}[\text{loop}(\text{B} \xrightarrow{\text{tt}} \text{C})] \text{ and } \forall \mathcal{A} : \emptyset \cdot \mathcal{A} = \mathcal{A}\}
\end{aligned}$$

$$\begin{aligned}
- F^2 &= \mathcal{F}^{ai} \llbracket \circ \rightarrow \begin{array}{c} \boxed{B} \xrightarrow{tt} \boxed{C} \\ \text{ff} \uparrow \downarrow \end{array} \circ \rrbracket (F^1) = \{\varepsilon\} \cup \{\varepsilon\} \cdot \{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket \\
&\quad \wr \text{definition of iteration and } \mathcal{F}^{ai} \llbracket \circ \rightarrow \begin{array}{c} \boxed{B} \xrightarrow{tt} \boxed{C} \\ \text{ff} \uparrow \downarrow \end{array} \circ \rrbracket \wr \\
&= \{\varepsilon\} \cup \{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket \quad \wr \text{since } \forall \mathcal{A} : \{\varepsilon\} \cdot \mathcal{A} = \mathcal{A} \wr \\
&= (\{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^0 \cup (\{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^1 \\
&\quad \wr \text{definition of natural powers } \mathcal{A}^0 \triangleq \{\varepsilon\} \text{ and } \mathcal{A}^1 \triangleq \mathcal{A} \wr
\end{aligned}$$

$$- F^n = \bigcup_{i=0}^{n-1} (\{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^i \quad \wr \text{induction hypothesis}$$

$$\begin{aligned}
- F^{n+1} &= \mathcal{F}^{ai} \llbracket \circ \rightarrow \begin{array}{c} \boxed{B} \xrightarrow{tt} \boxed{C} \\ \text{ff} \uparrow \downarrow \end{array} \circ \rrbracket (F^n) \quad \wr \text{definition of iterates} \\
&= \{\varepsilon\} \cup \left(\bigcup_{i=0}^{n-1} (\{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^i \right) \cdot \{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket \\
&\quad \wr \text{definition of } \mathcal{F}^{ai} \llbracket \circ \rightarrow \begin{array}{c} \boxed{B} \xrightarrow{tt} \boxed{C} \\ \text{ff} \uparrow \downarrow \end{array} \circ \rrbracket \text{ and induction hypothesis}
\end{aligned}$$

$$= (\{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^0 \cup \bigcup_{i=0}^{n-1} ((\{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^i \cdot \{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)$$

\wr definition of natural powers $\mathcal{A}^0 = \{\varepsilon\}$ and $\bigcup_{i \in \delta} (\mathcal{A}^i) \cdot \mathcal{A} = \bigcup_{i \in \delta} (\mathcal{A}^i \cdot \mathcal{A}) \wr$

$$= \bigcup_{i=0}^{(n+1)-1} (\{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^i$$

\wr definition of natural powers $\mathcal{A}^{n+1} = \mathcal{A}^n \cdot \mathcal{A}$ and definition of \cup

$$- \forall n \in \mathbb{N} : F^n = \bigcup_{i=0}^{n-1} (\{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^i \quad \wr \text{by recurrence}$$

$$\begin{aligned}
&\text{lfp}^{\subseteq} \mathcal{F}^{ai} \llbracket \circ \rightarrow \begin{array}{c} \boxed{B} \xrightarrow{tt} \boxed{C} \\ \text{ff} \uparrow \downarrow \end{array} \circ \rrbracket \\
&= \bigcup_{n \in \mathbb{N}} F^n \quad \wr \text{since } \mathcal{F}^{ai} \llbracket \circ \rightarrow \begin{array}{c} \boxed{B} \xrightarrow{tt} \boxed{C} \\ \text{ff} \uparrow \downarrow \end{array} \circ \rrbracket \text{ preserves unions} \\
&= \bigcup_{n \in \mathbb{N}} (\{B\} \cdot \mathcal{G}^{a*} \llbracket \circ \rightarrow \boxed{C} \rightarrow \circ \rrbracket)^n \quad \wr \text{definition of natural powers } \mathcal{A}^n \text{ and } \cup \wr
\end{aligned}$$

which we can write as $(\{B\} \cdot \mathcal{G}^{a*}[\circ \rightarrow \boxed{C} \rightarrow \circ])^*$.

– **Iteration, finite executions**

$$\begin{aligned}
 & \alpha^a(\mathcal{S}^{t*}[\text{while } (B) \{C\}]) \\
 = & \alpha^a(\text{let } \mathcal{S}^{ti}[\text{while } (B) \{C\}] \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while } (B) \{C\}] \text{ in } \{\pi \xrightarrow{A} \langle \text{while } (B) \{C\}, \rho \rangle \xrightarrow{\neg B} \\
 & \langle \text{stop}, \rho \rangle \mid \pi \in \Pi^* \wedge \pi \xrightarrow{A} \langle \text{while } (B) \{C\}, \rho \rangle \in \mathcal{S}^{ti}[\text{while } (B) \{C\}] \wedge \text{false} \in \mathcal{E}[\![B]\!]\rho\}) \\
 & \quad \quad \quad \wr \text{definition of } \mathcal{S}^{t*}[\text{while } (B) \{C\}]\wr \\
 = & \text{let } \mathcal{S}^{ti}[\text{while } (B) \{C\}] \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while } (B) \{C\}] \text{ in } \alpha^a(\{\pi \xrightarrow{A} \langle \text{while } (B) \{C\}, \rho \rangle \xrightarrow{\neg B} \\
 & \langle \text{stop}, \rho \rangle \mid \pi \in \Pi^* \wedge \pi \xrightarrow{A} \langle \text{while } (B) \{C\}, \rho \rangle \in \mathcal{S}^{ti}[\text{while } (B) \{C\}] \wedge \text{false} \in \mathcal{E}[\![B]\!]\rho\}) \\
 & \quad \quad \quad \wr \text{since } f(\text{let } \dots \text{ in } \dots) = \text{let } \dots \text{ in } f(\dots)\wr \\
 = & \text{let } \mathcal{S}^{ti}[\text{while } (B) \{C\}] \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while } (B) \{C\}] \text{ in } \alpha^a(\mathcal{S}^{ti}[\text{while } (B) \{C\}]) \cdot \neg B \\
 & \quad \quad \quad \wr \text{definition of } \alpha^a \text{ and ignoring the result } \text{false} \in \mathcal{E}[\![B]\!]\rho \text{ of test}\wr \\
 = & \alpha^a(\text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while } (B) \{C\}]) \cdot \neg B \quad \quad \quad \wr \text{definition of let } \dots \text{ in } \dots\wr \\
 \subseteq & \text{lfp}^{\subseteq} \mathcal{F}^{ai}[\circ \rightarrow \boxed{B} \xrightarrow{tt} \boxed{C} \rightarrow \circ] \cdot \neg B \\
 & \quad \quad \quad \wr \text{by fixpoint approximation and } \cdot \text{ preserves unions so is } \subseteq\text{-increasing}\wr \\
 = & \mathcal{G}^{a*}[\circ \rightarrow \boxed{B} \xrightarrow{tt} \boxed{C} \rightarrow \circ] \quad \quad \quad \wr \text{definition of } \mathcal{G}^{a*}[\circ \rightarrow \boxed{B} \xrightarrow{tt} \boxed{C} \rightarrow \circ]\wr
 \end{aligned}$$

– **Limit overapproximation** $\alpha^a(\text{lim}^t \mathcal{S}) = \text{lim}^a \alpha^a(\mathcal{S})$

$$\begin{aligned}
 & \alpha^a(\text{lim}^t \mathcal{S}) \\
 = & \{\alpha^a(\pi) \mid \pi \in \Pi^\infty \wedge \forall \pi_1 \in \Pi^* : (\exists \pi_2 \in \Pi^\infty : \pi = \pi_1 \xrightarrow{A} \pi_2) \implies (\exists \pi_3 \in \Pi^* : \exists \pi_4 \in \Pi^\infty : \\
 & \pi = \pi_1 \xrightarrow{A} \pi_3 \xrightarrow{A'} \pi_4 \wedge \pi_1 \xrightarrow{A} \pi_3 \in \mathcal{S})\} \quad \quad \quad \wr \text{definition of } \alpha^a \text{ and } \text{lim}^t\wr \\
 = & \{\underline{\pi} \mid \langle \bar{\pi}, \underline{\pi} \rangle \in \Pi^\infty \wedge \forall \pi_1 \in \Pi^* : (\exists \pi_2 \in \Pi^\infty : \langle \bar{\pi}, \underline{\pi} \rangle = \pi_1 \xrightarrow{A} \pi_2) \implies (\exists \pi_3 \in \Pi^* : \exists \pi_4 \in \\
 & \Pi^\infty : \langle \bar{\pi}, \underline{\pi} \rangle = \pi_1 \xrightarrow{A} \pi_3 \xrightarrow{A'} \pi_4 \wedge \pi_1 \xrightarrow{A} \pi_3 \in \mathcal{S})\} \triangleq A \quad \quad \quad \wr \text{since } \alpha^a(\langle \bar{\pi}, \underline{\pi} \rangle) = \underline{\pi}\wr
 \end{aligned}$$

$$\subseteq \{\omega \in \mathbb{A}^\infty \mid \forall \omega_1 \in \mathbb{A}^* : (\exists \omega_2 \in \mathbb{A}^\infty : \omega = \omega_1 \cdot \omega_2) \implies (\exists \omega_3 \in \mathbb{A}^* : \exists \omega_4 \in \mathbb{A}^\infty : \omega = \omega_1 \cdot \omega_3 \cdot \omega_4 \wedge \omega_1 \cdot \omega_3 \in \mathcal{S})\} \triangleq \mathbf{B}$$

{Let $\underline{\pi} \in \mathbf{A}$, we must show that $\underline{\pi} \in \mathbf{B}$. Assume $\underline{\pi} = \omega_1 \cdot \mathbf{A} \cdot \omega_2$, then $\langle \bar{\pi}, \underline{\pi} \rangle = \langle \underline{\pi}_1, \omega_1 \rangle \xrightarrow{\mathbf{A}} \langle \underline{\pi}_2, \omega_2 \rangle = \pi_1 \xrightarrow{\mathbf{A}} \pi_2$. So by the definition of \mathbf{A} , $\exists \pi_3 \in \Pi^* : \exists \pi_4 \in \Pi^\infty : \langle \bar{\pi}, \underline{\pi} \rangle = \pi_1 \xrightarrow{\mathbf{A}} \pi_3 \xrightarrow{\mathbf{A}'} \pi_4 \wedge \pi_1 \xrightarrow{\mathbf{A}} \pi_3 \in \mathcal{S}$. So define $\omega_1 = \underline{\pi}_1 \mathbf{A}$, $\omega_3 = \underline{\pi}_3$ and $\omega_4 = \mathbf{A}' \underline{\pi}_4$, then $\underline{\pi} = \omega_1 \cdot \omega_3 \cdot \omega_4$ and $\alpha^a(\pi_1 \xrightarrow{\mathbf{A}} \pi_3) = \omega_1 \cdot \omega_3 \in \alpha^a(\mathcal{S})$ proving that $\underline{\pi} \in \mathbf{B}$.}

$$= \lim^a \alpha^a(\mathcal{S}) \quad \{\text{definition of } \lim^a\}$$

– Iteration, infinite executions

$$\alpha^a(\mathcal{S}^{t\infty} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket)$$

$$= \alpha^a(\text{let } \mathcal{S}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \text{ in}$$

$$\lim^t \{ \pi \xrightarrow{\mathbf{A}} \langle \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \in \mathcal{S}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \mid \text{true} \in \mathcal{E} \llbracket \mathbf{B} \rrbracket \rho \} \cup \{ \pi \xrightarrow{\mathbf{A}} \langle \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \xrightarrow{\mathbf{B}} \langle \mathbf{C}; \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \xrightarrow{\mathbf{A}'} \pi' \ ; \ \text{while}(\mathbf{B}) \{ \mathbf{C} \} \mid \pi \xrightarrow{\mathbf{A}} \langle \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \in \mathcal{S}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \wedge \text{true} \in \mathcal{E} \llbracket \mathbf{B} \rrbracket \rho \wedge \langle \mathbf{C}, \rho \rangle \xrightarrow{\mathbf{A}'} \pi' \in \mathcal{S}^{t\infty} \llbracket \mathbf{C} \rrbracket \}$$

{definition of $\mathcal{S}^{t\infty} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket$ }

$$= \text{let } \mathcal{S}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \text{ in}$$

$$\alpha^a(\lim^t \{ \pi \xrightarrow{\mathbf{A}} \langle \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \in \mathcal{S}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \mid \text{true} \in \mathcal{E} \llbracket \mathbf{B} \rrbracket \rho \} \cup \{ \pi \xrightarrow{\mathbf{A}} \langle \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \xrightarrow{\mathbf{B}} \langle \mathbf{C}; \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \xrightarrow{\mathbf{A}'} \pi' \ ; \ \text{while}(\mathbf{B}) \{ \mathbf{C} \} \mid \pi \xrightarrow{\mathbf{A}} \langle \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \in \mathcal{S}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \wedge \text{true} \in \mathcal{E} \llbracket \mathbf{B} \rrbracket \rho \wedge \langle \mathbf{C}, \rho \rangle \xrightarrow{\mathbf{A}'} \pi' \in \mathcal{S}^{t\infty} \llbracket \mathbf{C} \rrbracket \}$$

{since $f(\text{let } \dots \text{ in } \dots) = \text{let } \dots \text{ in } f(\dots)$ }

$$\subseteq \text{let } \mathcal{S}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \text{ in}$$

$$\lim^a \alpha^a(\{ \pi \xrightarrow{\mathbf{A}} \langle \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \in \mathcal{S}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \mid \text{true} \}) \cup \alpha^a(\{ \pi \xrightarrow{\mathbf{A}} \langle \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \xrightarrow{\mathbf{B}} \langle \mathbf{C}; \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \xrightarrow{\mathbf{A}'} \pi' \ ; \ \text{while}(\mathbf{B}) \{ \mathbf{C} \} \mid \pi \xrightarrow{\mathbf{A}} \langle \text{while}(\mathbf{B}) \{ \mathbf{C} \}, \rho \rangle \in \mathcal{S}^{ti} \llbracket \text{while}(\mathbf{B}) \{ \mathbf{C} \} \rrbracket \wedge \langle \mathbf{C}, \rho \rangle \xrightarrow{\mathbf{A}'} \pi' \in \mathcal{S}^{t\infty} \llbracket \mathbf{C} \rrbracket \}$$

$$\begin{aligned}
& \wr \text{by limit approximation of } \alpha^a \text{ which preserves unions, and ignoring the result} \\
& \text{true} \in \mathcal{E}[\mathbb{B}]\rho \text{ of the test} \wr \\
= & \text{let } \mathcal{S}^{ti}[\text{while}(\mathbb{B})\{C\}] \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while}(\mathbb{B})\{C\}] \text{ in } \lim^a \alpha^a(\mathcal{S}^{ti}[\text{while}(\mathbb{B})\{C\}]) \cup \\
& \{\alpha^a(\pi_1) \cdot \mathbb{B} \cdot \alpha^a(\langle C, \rho \rangle \xrightarrow{A'} \pi') \mid \pi_1 \in \mathcal{S}^{ti}[\text{while}(\mathbb{B})\{C\}] \wedge \langle C, \rho \rangle \xrightarrow{A'} \pi' \in \mathcal{S}^{t\infty}[C]\} \\
& \wr \text{since } \{x \in S \mid \text{true}\} = S, \text{ letting } \pi_1 = \pi \xrightarrow{A} \langle \text{while}(\mathbb{B})\{C\}, \rho \rangle, \alpha^a(\pi \xrightarrow{A} \pi') = \\
& \alpha^a(\pi) \cdot A \cdot \alpha^a(\pi') \text{ and } \alpha^a(\pi \text{ ; } C) = \alpha^a(\pi) \wr \\
= & \text{let } \mathcal{S}^{ti}[\text{while}(\mathbb{B})\{C\}] \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while}(\mathbb{B})\{C\}] \text{ in } \lim^a \alpha^a(\mathcal{S}^{ti}[\text{while}(\mathbb{B})\{C\}]) \cup \\
& \{\alpha^a(\pi_1) \cdot \mathbb{B} \cdot \alpha^a(\pi_2) \mid \pi_1 \in \mathcal{S}^{ti}[\text{while}(\mathbb{B})\{C\}] \wedge \pi_2 \in \mathcal{S}^{t\infty}[C]\} \\
& \wr \text{letting } \pi_2 = \langle C, \rho \rangle \xrightarrow{A'} \pi' \wr \\
= & \text{let } \mathcal{S}^{ti}[\text{while}(\mathbb{B})\{C\}] \triangleq \text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while}(\mathbb{B})\{C\}] \text{ in } \lim^a \alpha^a(\mathcal{S}^{ti}[\text{while}(\mathbb{B})\{C\}]) \cup \\
& \alpha^a(\mathcal{S}^{ti}[\text{while}(\mathbb{B})\{C\}]) \cdot \{\mathbb{B}\} \cdot \alpha^a(\mathcal{S}^{t\infty}[C]) \\
& \wr \text{since } \{\alpha^a(\pi) \cdot \alpha^a(\pi') \mid \pi \in S \wedge \pi' \in S'\} = \alpha^a(S) \cdot \alpha^a(S') \wr \\
= & \lim^a \alpha^a(\text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while}(\mathbb{B})\{C\}]) \cup \alpha^a(\text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while}(\mathbb{B})\{C\}]) \cdot \{\mathbb{B}\} \cdot \alpha^a(\mathcal{S}^{t\infty}[C]) \\
& \wr \text{definition of let ... in ...} \wr \\
\subseteq & \lim^a \alpha^a(\text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while}(\mathbb{B})\{C\}]) \cup \alpha^a(\text{lfp}^{\subseteq} \mathcal{F}^{ti}[\text{while}(\mathbb{B})\{C\}]) \cdot \{\mathbb{B}\} \cdot \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C} \rightarrow \circ] \\
& \wr \text{by induction hypothesis } \alpha^a(\mathcal{S}^{t\infty}[C]) \subseteq \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C} \rightarrow \circ] \text{ and } \cdot \text{ is } \subseteq\text{-increasing} \wr \\
\subseteq & \lim^a \text{lfp}^{\subseteq} \mathcal{F}^{ai}[\circ \rightarrow \boxed{\mathbb{B}} \xrightarrow{\text{tt}} \boxed{C} \rightarrow \circ] \cup \text{lfp}^{\subseteq} \mathcal{F}^{ai}[\circ \rightarrow \boxed{\mathbb{B}} \xrightarrow{\text{tt}} \boxed{C} \rightarrow \circ] \cdot \{\mathbb{B}\} \cdot \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C} \rightarrow \circ] \\
& \wr \text{by fixpoint approximation, } \cdot \text{ and } \lim^a \text{ are } \subseteq\text{-increasing} \wr \\
= & \text{let } W = \text{lfp}^{\subseteq} \mathcal{F}^{ai}[\circ \rightarrow \boxed{\mathbb{B}} \xrightarrow{\text{tt}} \boxed{C} \rightarrow \circ] \text{ in } \lim^a W \cup W \cdot \{\mathbb{B}\} \cdot \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{C} \rightarrow \circ] \\
& \wr \text{since } f(e) = \text{let } x = e \text{ in } f(x) \wr \\
= & \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{\mathbb{B}} \xrightarrow{\text{tt}} \boxed{C} \rightarrow \circ] \wr \text{definition of } \mathcal{G}^{a\infty}[\circ \rightarrow \boxed{\mathbb{B}} \xrightarrow{\text{tt}} \boxed{C} \rightarrow \circ] \wr
\end{aligned}$$

Bibliography

- [BCD⁺11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.
- [BDH⁺13] M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening. An abstract interpretation of DPLL(T). In *Proc. of the conference on Verification, Model Checking and Abstract Interpretation*, 2013.
- [BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. A generalization of Shostak's method for combining decision procedures. In Alessandro Armando, editor, *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FroCoS '02)*, volume 2309 of *Lecture Notes in Artificial Intelligence*, pages 132–146. Springer-Verlag, April 2002. Santa Margherita Ligure, Italy.
- [BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Inter-*

pretation, *8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2007.

- [BHRZ05] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. *Sci. Comput. Program.*, 58(1-2):28–56, 2005.
- [BHZ07] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. *STTT*, 9(3-4):413–414, 2007.
- [BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What’s decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI’06*, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag.
- [BSST09] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.
- [BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at www.SMT-LIB.org.
- [BTV03] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *J. Autom. Reasoning*, 31(2):129–168, 2003.
- [CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

- [CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [CC79a] P. Cousot and R. Cousot. Constructive versions of Tarski’s fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.
- [CC79b] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
- [CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of *Journal of Logic Programming* has mistakenly published the unreadable galley proof. For a correct version of this paper, see <http://cs.nyu.edu/~pcousot/>).
- [CC92b] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
- [CC92c] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP ’92*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

- [CCF⁺07] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, pages 272–300, Tokyo, Japan, LNCS 4435, December 6–8 2007. Springer, Berlin.
- [CCM10a] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. Logical abstract domains and interpretations. In S. Nanz, editor, *The Future of Software Engineering*, pages 48–71. Springer-Verlag, Heidelberg, 2010.
- [CCM10b] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. A scalable segmented decision tree abstract domain. In Z. Manna and D. Peled, editors, *Pnueli Festschrift*, volume 6200 of *Lecture Notes in Computer Science*, pages 72–95, Heidelberg, 2010. Springer-Verlag.
- [CCM11] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In M. Hofmann, editor, *14th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2011), Saarbrücken, Germany*, volume 6604 of *Lecture Notes in Computer Science*, pages 456–472. Springer-Verlag, Heidelberg, March 26 – April 3, 2011.
- [CGG⁺05] Alexandru Costan, Stéphane Gaubert, Éric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 462–475. Springer, 2005.

- [CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.
- [CK90] C. Chang and H. Keisler. *Model theory*, volume 73 of *Studies in logic and the foundation of mathematics*. Elsevier Science, New York, NY, USA, third edition, 1990.
- [Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
- [Cou02] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.
- [CSS03] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003.
- [DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

- [dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
- [ea96] J. L. Lions et al. Ariane 5, flight 501 failure, report by the inquiry board, 1996. <http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html>.
- [GC10] Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2010.
- [GGTZ07] Stéphane Gaubert, Éric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007.
- [GHN⁺04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference,*

- CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.
- [GR98] Roberto Giacobazzi and Francesco Ranzato. Optimal domains for disjunctive abstract intepretation. *Sci. Comput. Program.*, 32(1-3):177–210, 1998.
- [GR07] Denis Gopan and Thomas W. Reps. Guided static analysis. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007.
- [GS07] Thomas Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2007.
- [Hal79] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. Thèse, Institut National Polytechnique de Grenoble - INPG ; Université Joseph-Fourier - Grenoble I, March 1979.
- [Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.
- [HMM12] Julien Henry, David Monniaux, and Matthieu Moy. PAGAI: a path sensitive static analyzer. *CoRR*, abs/1207.3937, 2012.

- [HT98] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS*, volume 1503 of *LNCS*, pages 200–214. Springer, 1998.
- [JM09] Bertrand Jeannet and Antoine Miné. APRON: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.
- [JP04] Kurt Jensen and Andreas Podelski, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.
- [Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.
- [Kar84] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.
- [Kil73] Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973.
- [Kle52] Stephen Cole Kleene. *Introduction to metamathematics*. Bibl. Matematica. North-Holland, Amsterdam, 1952.

- [LM05] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient decision procedure for UTVPI constraints. In Bernhard Grämlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2005.
- [LRR13] Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio. SMT-based array invariant generation. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2013.
- [MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005.
- [Mun64] J.R. Munkres. *Elementary Linear Algebra*. Addison-Wesley series in mathematics. Addison-Wesley, 1964.
- [NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
- [NO03] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In Moshe Y. Vardi and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 10th International Conference, LPAR*

2003, Almaty, Kazakhstan, September 22-26, 2003, *Proceedings*, volume 2850 of *Lecture Notes in Computer Science*, pages 78–90. Springer, 2003.

- [NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.
- [NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006.
- [Opp80] Derek C. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3):403–411, 1980.
- [Pap81] Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, October 1981.
- [RCK04] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, ISSAC '04*, pages 266–273, New York, NY, USA, 2004. ACM.
- [Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.
- [SBDL01] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, pages 29–37. IEEE Computer Society, 2001.

- [Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.
- [Seb07] Roberto Sebastiani. Lazy satisfiability modulo theories. *JSAT*, 3(3-4):141–224, 2007.
- [SISG06] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2006.
- [SJ80] Norihisa Suzuki and David Jefferson. Verification decidability of presburger array programs. *J. ACM*, 27(1):191–205, 1980.
- [Ske92] R. Skeel. Roundoff error and the patriot missile, November 1992. <http://www.siam.org/siamnews/general/patriot.htm>.
- [SW04] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In Jensen and Podelski [JP04], pages 280–295.
- [Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pac. J. Math.*, 5:285–309, 1955.
- [TR12] Aditya V. Thakur and Thomas W. Reps. A method for symbolic computation of abstract operations. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 174–192. Springer, 2012.

- [UM14] Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014.
- [Ver94] H. Le Verge. A note on chernikova’s algorithm. Technical report, 1994.
- [WZH⁺13] Kirsten Winter, Chenyi Zhang, Ian J. Hayes, Nathan Keynes, Cristina Cifuentes, and Lian Li. Path-sensitive data flow analysis simplified. In Lindsay Groves and Jing Sun, editors, *Formal Methods and Software Engineering - 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 - November 1, 2013, Proceedings*, volume 8144 of *Lecture Notes in Computer Science*, pages 415–430. Springer, 2013.
- [YRS04] Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In Jensen and Podelski [JP04], pages 530–545.