TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Informatik 7

Constraint Solving for Verification

Ashutosh Kumar Gupta

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Helmut Seidl
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Andrey Rybalchenko
	2.	Full Prof. Dr. Rupak Majumdar, University of California/ USA

Die Dissertation wurde am 30.05.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.07.2011 angenommen.

Abstract

Software is widely used and hard to make reliable. Researchers have been exploring new ways to ensure software reliability including software verification, i.e., mathematical reasoning about software. The current technology for software verification is not sufficiently efficient to be used in industrial software production. In this thesis, we present novel constraint based verification methods and algorithms for constraint solving that increase the efficiency of software verification.

In the direction of constraint based verification methods, we first present an algorithm that improves the efficiency of an important verification method, namely template based invariant generation [16]. Then, we extend the template based invariant generation method to compute bounds on consumption of a resource by a program. In particular, we apply our bound computation algorithm on computing bounds of heap consumption of C programs.

In the direction of algorithms for constraint solving, we first present a novel simplex based proof production algorithm that is compatible with the simplex algorithm employed in CLP(Q) [52]. Secondly, we present algorithm for solving recursion-free Horn clauses over LI+UIF. We use these algorithms for refinement procedures in model-checkers to verify multi-threaded programs, programs with procedures, and higher order functional programs.

Acknowledgments

First and foremost, I am very grateful to my PhD supervisor Prof. Andrey Rybalchenko, whose guidance was very productive for my development as a researcher. I also thank Prof. Rupak Majumdar for reading this thesis and giving valuable comments.

I am thankful to Prof. Thomas Henzinger for teaching me course computer aided verification. The course introduced me to the formal methods and verification. I found formal methods very challenging and intellectually satisfying for their systematic view of computing. The encouragement of Nir Piterman as a TA of the course also contributed in my final decision to pursue PhD in verification. I thank to the researchers with whom I collaborated for research work: Corneliu Popeea, Ru-Gang Xu, Satnam Singh, Jiri Simsa, Viktor Vafeiadis, Byron Cook, and Stephan Magill. Specially, I thank to Corneliu Popeea for working with me on a long running project. It was wonderful experience to work with him.

I was affiliated as a PhD student at EPFL, MPI-SWS, and TUMunich during the course of my PhD. I enjoyed working at all the institutes and able to gather a wide experience of working at different kinds of institutes in Europe. I would also like to thank my colleagues during my PhD: Ruslan, Juan, Andreas, Jan, Atul, Animesh, Alan, Nuno, Mia, Max, Bimal, Malveeka, Pedro, Surender, Simon, Dietmar, Maria, Barbara, and Vasu. They all made the institutes wonderful places to be, and being in Europe is an experience that I will always treasure.

I would also like to thank my long term friends now living around the world: Prakash, Mamta, Chaitanya, Manish, Vertika, Ajeeta, Vaibhav, Bramhdatt, Michael, Florance, Saad, Adnan, Yogesh, and Sudhir. Occasional conversation with them always gives me a great joy.

I would like to thank my wife Divya. Her love and encouragement was vital for my success. Finally, I would like to express my deep gratitude to my parents, my sister, and my brother-in-law for their love and support during the ups and downs of graduate school. I am grateful beyond words for all that they have given me.

Contents

Chapter 1

Introduction

Software is widely used. A personal computer may be executing millions of lines of source code to process the complex interactions of different components of the computer. Designing reliable software is very hard due to its high complexity and size. Currently, engineers in the software industry are applying many techniques to increase the reliability of software, e.g., testing and code review. However, these techniques have limitations, hence researchers have been exploring new ways to ensure software reliability. For example, mathematical reasoning about software, which is known as software verification, has the potential to improve software reliability. The methods for software verification are designed by composing the algorithms for constraint solving as building blocks.

Current technology for software verification is not sufficiently efficient to apply to large programs. In this thesis, we will present novel constraint based verification methods and algorithms for constraint solving that increase the efficiency of software verification. We only consider safety verification for programs that are constructed using updates and condition expressions that are represented using conjunction of linear (in)equalities. Along with this thesis, this class of programs has been focus of a large body of research because many software applications are in this class and mathematical properties of the linear operations leads to the development of verification methods with practical computation complexities.

Our contribution is separated in the two parts: constraint based verification methods and algorithms for constraint solving.

Part I : Constraint based verification methods

In this part, we first present an algorithm that improves the efficiency of an important verification method, namely template based invariant generation [16]. Then, we extend the template based invariant generation method to compute bounds on consumption of a resource by a program. In particular, we apply our bound computation algorithm on computing bounds of heap consumption of C programs.

From tests to proofs: We first present an algorithm that improves the efficiency of template based invariant generation [16]. An invariant of a program is a super set of the reachable program states. Templates are assertions over the program variables and parameters. By choosing values for the template parameters, we select an assertion over program variables. Here, templates are used to represent the unknown invariants. Using the program and a template, constraints are generated whose solutions are the invariants. The generated constraints are non-linear, which are hard to solve.

Our algorithm is a heuristic approach that accelerates the non-linear constraint solving by taking advantage of executions of the program. First, our algorithm collects test executions for the program. The program states visited by the test executions must satisfy every correct instantiation of the invariant templates since reachable states must satisfy the instantiation of the invariant templates. For each program state visited by test executions, we replace program variables occurring in template by their values as determined by the program state and obtain simpler constraints over template parameters. Then, we collect these linear constraints for each visited program state. We conjoin these linear constraints with the original non-linear constraints and solve the result. The additional linear constraints are helpful in guiding a constraint solver towards a solution of the non-linear constraints. As a result, the constraints are solved faster.

We designed and implemented a tool called InvGen that implements this heuristic and tested InvGen on benchmarks. Results of the tests show that this heuristic is helpful for many examples. If the heuristic fails to help then it does not slow down the overall solving either. We applied InvGen for computing 'path invariants' [5] for counter examples in a CEGAR based tool Blast [45]. InvGen helps Blast to terminate for some examples on which Blast did not terminate without InvGen.

Bound Synthesis: We extend template based invariant generation [16] to compute bounds on consumption of a program resource, e.g., time, memory, or network bandwidth. We add an auxiliary variable and updates on the variable to represent consumption of the resource. Together with an invariant template for each program location, our bound synthesis method also assumes a template that expresses a space of linear upper bounds over other program variables. Using these templates, we apply the algorithm of template based invariant generation. A solution of the templates provides the symbolic expressions that bound the consumption variable. We implemented this technique in a tool BoundGen.

C-to-Gates synthesis using BoundGen: We applied our bound synthesis algorithm for computing bounds of heap consumption of C programs. We instantiated the resource as heap in the bound synthesis algorithm. We combined BoundGen with a shape analysis tool Thor [63] and a hardware synthesis toolchain. Using this combined toolchain, we directly synthesized hardware circuits from C programs that allocate memory dynamically.

Part II : Constraint solving algorithms

In this part, we first present a novel simplex based proof production algorithm that is compatible with the simplex algorithm employed in CLP(Q) [52]. Secondly, we present algorithm for solving recursion-free Horn clauses over LI+UIF. We use these algorithms for refinement procedures in model-checkers to verify multi-threaded programs, programs with procedures, and higher order functional programs.

Proof producing CLP(LI+UIF): For many path based constraint generation and solving methods of verification, the significant cost of verification goes into solving constraints obtained from symbolic execution of program paths. The constraints are solved by interpolation procedures [66]. An efficient interpolation procedure can reduce the verification time. We use CLP(Q) [52] for manipulating and solving constraints, which is a simplex based tool. We instrument CLP(Q) in order to compute interpolants. CLP(Q) requires eager equality propagation, which is beneficial for dealing with interpolation queries for program paths, since they may contain a large number of variables together with a large number of equality constraints between them. The existing simplex based interpolation algorithms [14] require a proof of the unsatisfiability from the simplex. The proof producing algorithms [14, 23] instrument input constraints, which adds many slack variables, and forbid equality propagation in the employed simplex algorithm, which increases cost of equality detection. Therefore, these algorithms are sub-optimal for our application. We address this deficiency by developing a variation of simplex based proof producing algorithm that maintains additional information required for proof production alongside the simplex tableau.

Programs that involve non-linear arithmetic operations can be verified by approximating these operations using uninterpreted functions. We also designed and implemented a tool CLP(LI+UIF) that can find contradiction in formulas in theory of linear arithmetic with uninterpreted function symbols and returns a proof tree for the found contradiction. Using this proof tree and algorithms in [65, 72], we compute interpolants efficiently.

Solving recursion-free Horn clauses over LI+UIF: The path constraints obtained from multi-threaded programs, program with procedures, and higher order functional programs are sets of Horn clauses. We have developed an algorithm for solving Horn clauses over linear arithmetic with uninterpreted function symbols. Our algorithm extends [65, 72] by taking branching structure of Horn clauses into account. We have designed and implemented a tool based on this algorithm. Using this tool, we have designed and implemented model-checkers for multi-threaded programs, programs with procedures, and higher order functional programs.

Contributions

This thesis makes the following contributions.

- An algorithm for efficient template based invariant generation using test data.
- Design and implementation of the InvGen tool that implements the above algorithm.
- A template based algorithm for resource bound synthesis.
- A tool for C-to-Gates synthesis using the resource bound synthesis algorithm.
- Design and implementation of a proof producing CLP(LI+UIF).
- An algorithm for solving recursion-free Horn clauses over LI+UIF.
- Design and implementation of model checkers for multi-threaded programs, programs with procedures, and higher order functional programs using the above Horn clauses solving algorithm.

Chapter 3 is based on [36, 40]. Chapters 4 and 5 are based on [17]. Chapters 7 is under submission. Chapter 8 is based on [38]. A refinement tool based on chapter 8 is used for [37, 39].

Chapter 2

Basic notation and programs

In this chapter, we describe the mathematical notation used in this thesis.

2.1 Basic notation

Let $\mathbb{N}$ be the set of natural numbers. For $i, j \in \mathbb{N}$, let $i..j = \{x | i \leq x \leq j\}$. Let $\mathbb{Q}$ be the set of rational numbers. We use the standard definition of relations $<$, $\leq$, and $=$ over $\mathbb{N}$ and $\mathbb{Q}$. Let $+\infty$ and $-\infty$ be positive and negative infinity respectively. We extend $<$ to $\mathbb{Q} \cup \{+\infty, -\infty\}$ in following way. Let $c \in \mathbb{Q}$.

$$
\begin{array}{llll}
-\infty < c := \mathit{true} & c < +\infty := \mathit{true} & -\infty < +\infty := \mathit{true} & +\infty < +\infty := \text{undefined} \\
+\infty < c := \mathit{false} & c < -\infty := \mathit{false} & +\infty < -\infty := \mathit{false} & -\infty < -\infty := \text{undefined}
\end{array}
$$

We also extend arithmetic operation as follows $(-\infty) + c = -\infty$, $(+\infty) + c = +\infty$, $(+\infty) - (-\infty) = +\infty$, $(-\infty) - (+\infty) = -\infty$, $(+\infty) - (+\infty) = \text{undefined}$, and $(-\infty) - (-\infty) = \text{undefined}$.

Let *sequence* be an abstract data type. Let $\bullet$ be an operator that contact two sequences or a sequence and an element. Let $\_$ denote an existentially quantified anonymous variable in a formula.

2.2 Theory of linear arithmetic and uninterpreted functions

This section presents the syntax and semantics of the theory of linear arithmetic and uninterpreted functions. Let $\mathcal{T}_{\text{LI+UIF}}$ denote this theory.

Syntax

We assume countable sets of *variables* X, with $x \in X$, and *function symbols* $\mathcal{F}$, with $f \in \mathcal{F}$. Let the arity of function symbols be encoded in their names. In addition, we assume a set of *rational numbers* $\mathbb{Q}$, with $\{0, c\} \subseteq \mathbb{Q}$, and an inequality symbol $\leq$. Following grammar defines a quantifier-free class of formulas in $\mathcal{T}_{\text{LI+UIF}}$.

$$
\begin{array}{llll}
\text{terms} \ni t & ::= c \mid x \mid ct \mid t + t \mid f(t, \dots, t) & \text{conjunctive constraints} \ni C & ::= A \mid C \wedge C \\
\text{atoms} \ni A & ::= t \leq 0 & \text{constraints} \qquad \ni F & ::= A \mid \neg F \mid F \wedge F \mid F \vee F
\end{array}
$$

Semantics

For abbreviation, let $\models$ denote $\models_{\mathcal{T}_{\text{LI+UIF}}}$ in the part II of the thesis. Then, a constraint F is valid if it is satisfied by every assignment of its free variables with rational numbers. We write $\models F$ when F is valid.

Auxiliary definitions

Let *subterms*(F) be the subterms occurring in a constraint F and *atoms*(F) be the atoms occurring in F. We assume that t is in a minimized form when $Smb(t)$ is evaluated. For example, $Smb(x + y - x) = \{y\}$.

2.3 Program

We assume an abstract representation of programs by transition systems [64]. A *program* $\mathcal{P} = (V, \mathcal{L}, \ell_{init}, \mathcal{T}, \ell_{err})$ consists of a set V of variables, a set $\mathcal{L}$ of control locations, an initial location $\ell_{init} \in \mathcal{L}$, a set $\mathcal{T}$ of transitions, and an error location $\ell_{err} \in \mathcal{L}$. Each transition $\tau \in \mathcal{T}$ is a tuple (ℓ, ρ, ℓ'), where $\ell, \ell' \in \mathcal{L}$ are control locations, and ρ is a constraint over variables from $V \cup V'$. The variables from V denote values at control location ℓ, and the variables from V' denote the values of the variables from V at control location ℓ'. The error location ℓ_{err} is used to represent assertion statements. Each failed assertion leads to ℓ_{err}. We assume that the error location ℓ_{err} does not have any outgoing transitions. The sets of locations and transitions naturally define a directed graph, called the *control flow graph* (CFG) of the program, which puts the transition constraints at the edges of the graph.

A *state* of the program $\mathcal{P}$ is a valuation of the variables V. We shall represent sets and binary relations over states using constraints over V and V' in the standard way. A *computation* of $\mathcal{P}$ is a sequence of location and state pairs $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots$ such that ℓ_0 is the initial location and for each consecutive $\langle \ell_i, s_i \rangle$ and $\langle \ell_{i+1}, s_{i+1} \rangle$ there is a transition $(\ell_i, \rho, \ell_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$. A state s is *reachable* at location ℓ if $\langle \ell, s \rangle$ appears in some computation. The program is *safe* if the error location ℓ_{err} does not appear in any computation. A *path* of the program $\mathcal{P}$ is a finite or infinite sequence $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), \ldots$ of transitions, where ℓ_0 is the initial location. The path π is *feasible* if there is a computation $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots$ such that each consecutive pair of states (s_i, s_{i+1}) is induced by the corresponding transition, i.e., $(s_i, s_{i+1}) \models \rho_i$, otherwise π is called *infeasible*. A path that ends at the error location is called an *error path* (or *counterexample path*).

An *invariant* of $\mathcal{P}$ at a location $\ell \in \mathcal{L}$ is a super-set of states that are reachable at ℓ, which we represent by an assertion over V. An *inductive invariant map* assigns an invariant to each program location such that for each transition $(\ell, \rho, \ell') \in \mathcal{T}$ the implication $\eta(\ell) \wedge \rho \to (\eta(\ell'))'$ is valid, where $(\eta(\ell'))'$ is the assertion obtained by substituting variables V with the variables V' in $\eta(\ell')$. We observe that due to the invariance condition we have $\eta(\ell_{init}) = \mathit{true}$. An invariant map is *safe* if it assigns an empty set to the error location, i.e., $\eta(\ell_{err}) = \mathit{false}$.

A safe inductive invariant map serves as a proof that the error location cannot be reached on any program execution, and hence that the program is safe. The *invariant-synthesis* problem is to construct such a map for a given program.

Some program analysis techniques solve invariant synthesis problem by computing super set of reachable states along an (in)feasible path at a time. For an finite infeasible path $\pi = (\ell_0, \rho_0, \ell_1), \ldots, (\ell_n, \rho_n, \ell_{n+1})$, let $I_1, \ldots, I_{n-1}$ be a sequence assertions over V that satisfy

$$\mathit{true} \wedge \rho_0 \to I_1' \qquad I_1 \wedge \rho_1 \to I_2' \quad \ldots \quad I_{n-1} \wedge \rho_{n-1} \to I_n' \qquad I_n \wedge \rho_n \to \mathit{false}.$$

The above constraints are known as *interpolation constraints* for π. By solving the interpolation constraints, we can compute the super set of reachable states along π. The problem of solving interpolation constraints is easier than invariant-synthesis problem because there are no circular dependencies between unknown assertions.

Part I

Constraint based verification methods

Chapter 3

From tests to proofs

Programmers make mistakes, and much time and effort is spent on finding and fixing these mistakes. While it has long been known that *program invariants* are the key to proving a program correct with respect to a safety property [27, 48], their applicability has been limited in practice since they often require explicit and expensive programmer annotations. To circumvent this problem, there has been considerable research effort in program analysis for *automatic* inference of program invariants [2, 4, 7, 47, 76]. In these algorithms, a set of constraints is generated from the program text whose solution provides an inductive invariant proof of program correctness.

In the *abstract interpretation* based approach [7, 19, 67] to inductive invariant inference, one computes the fixpoint of the program semantics relative to an abstract domain. In case the abstract domain has infinite height (for example, the domain of polyhedra), termination of the fixpoint computation is enforced by a widening operator. In the *counterexample-guided abstraction refinement (CEGAR)* approach [2, 47], one starts with a set of predicates, and uses spurious counterexamples produced by model checking to dynamically discover new predicates that serve as building blocks for the proof of program correctness. Finally, in the *constraint-based approach* [16, 35, 76], a parametric representation of an invariant map serves a starting point. Then, inductiveness and safety conditions are encoded as constraints on the parameters. Once these constraints have been determined, any satisfying assignment is guaranteed to yield an inductive invariant of the program. For example, an invariant template in linear arithmetic will specify for each program point an expression of the form $\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n \leq 0$, where $x_1, \ldots, x_n$ are program variables, and $\alpha_0, \ldots, \alpha_n$ are unknown parameters. The control flow graph of the program will specify constraints on the parameters at each program point, such that a global solution for all the α's produces an invariant.

While these techniques hold the potential for extremely sophisticated reasoning about programs, each

File	State-of-the-art techniques				This chapter
	InterProc	Blast	InvGen	InvGen+Z3	
Seq	$\times$	diverge	23s	1s	**0.5s**
Seq-z3	$\times$	diverge	23s	9s	**0.5s**
Seq-len	$\times$	diverge	T/O	T/O	**2.8s**
nested	$\times$	**1.2s**	T/O	T/O	2.3s
svd(light)	$\times$	50s	T/O	T/O	**14.2s**
heapsort	$\times$	**3.4s**	T/O	T/O	13.3s
mergesort	$\times$	**18s**	T/O	52s	170s
SpamAssassin-loop*	✓	22s	T/O	5s	**0.4s**
apache-get-tag*	$\times$	5s	**0.4s**	10s	0.7s
sendmail-fromqp*	$\times$	diverge	0.3s	5s	**0.3s**

Table 3.1: Comparison of invariant-based verification tools on benchmark problems.

technique by itself often fails to verify programs, since in practice reasoning about correctness often requires combining the strength of each individual approach. In this chapter, we demonstrate the potential of such a combination. We describe the design and implementation of a constraint-based invariant generator for linear arithmetic invariants. In our implementation, we use information from static abstract interpretation-based techniques as well as from dynamic testing to aggressively simplify constraints. Our experimental results demonstrate that using these optimizations our invariant generator can automatically verify many problems for which all the existing approaches we tried are unsuccessful.

It is important to mention that for each of our examples there is (in theory) a polyhedral abstract domain equipped with a suitable widening operator that can successfully prove the desired assertion. Our approach targets the cases for which the *existing* abstract interpreters fail due to heuristic choices made in the implementation that trade off precision for speed. For example, Figure 3.1(a) shows a program from [33] for which an abstract interpreter implementing the standard convex hull-based widening cannot prove the assertion. In our experiments, the abstract interpretation tool InterProc finds the invariants $\mathtt{z} = 10\mathtt{w}$ and $\mathtt{y} \leq 100\mathtt{x}$ at line 2 but not the crucial $\mathtt{y} \geq \mathtt{x}$. We observed that our approach finds the missing fact $\mathtt{y} \geq \mathtt{x}$ which together with the invariants found by InterProc, is sufficient to prove the assertion.

Table 3.1 shows the results of running a collection of state-of-the-art program verification tools on a set of common benchmark programs for software verification, including some challenge programs from [59], which are marked with the star symbol "*". InterProc [60] is a tool based on abstract interpretation (we used the PPL library together with the octagon domain when applying InterProc). Blast [47] is a software model checker based on counterexample refinement. InvGen is our previous implementation of constraint-based invariant generation using constraint logic programming (CLP) as a constraint solver [4]. InvGen+Z3 is the same constraint-based invariant generator but using the Z3 decision procedure [21] as the constraint solver, which applies the Boolean satisfiability-based encoding proposed in [35]. As is evident from Table 3.1, the results we obtained for the existing tools on the benchmark examples are disappointing. In Column 2, there is a "×" mark for each program for which InterProc was too imprecise to verify the assertion. In Column 3, the counterexample refinement procedure of Blast diverges on several examples. In Columns 4 and 5, the invariant generation procedures time out, denoted by "T/O", on most examples as the constraints become too hard to solve (both for CLP and for SAT). In contrast, our technique is able to efficiently solve all the examples, as shown in the last column.

While our invariant generator can be used in isolation, we have also integrated it with the Blast software model checker and have used it as the counterexample refinement engine using path programs [5]. Invariants for path programs provide additional predicates that refine the abstraction for the software model checker, and can produce better refinement predicates than usually available with current techniques, e.g. [46]. Software model checkers with path program-based counterexample analysis are well-suited for our techniques because they (automatically) generate small program units to either test for bugs or provide invariants. Using this integration, we have applied our implementation to verify a set of software verification benchmark programs [59] recently introduced as a challenge to the community. The examples in the benchmark set are extracted from common security-critical code, and contain assertions related to buffer bounds checking. Our implementation was able to verify all the (correct) programs in the benchmark in about 10s of total time.

Related Work This chapter is based on the conference version [36] and extends it with a directed symbolic execution technique that supports the dynamic strengthening, see Section 3.4, and a description of the InvGen tool, see Section 3.5.

Our work is influenced by recent advances in automatic static inference of inductive invariants using constraint solving [18, 35, 75] as well as by the use of dynamic analysis to estimate and infer likely system properties [24].

Constraint-based invariant synthesis techniques using templates in linear [4,16,35] and polynomial [58,75] arithmetic have been extensively studied, but their application has been limited by the cost of the constraint solving process. As we demonstrate in our experiments, even on quite small examples the constraint solver is likely to time-out. Our static and dynamic constraint simplification techniques limit the search space for the constraint solvers. Our experiments demonstrate orders of magnitude improvements over existing making it feasible to apply these techniques to larger programs.

```
int x=0; y=0; z=0 w=0;
while(*){
 if(*){
   x++; y+=100;
 }else if(*){
   if (x>=4){ x++; y++; }
 }else if(y>10*w && z>=100*x){
   y=-y;
 }
 w++; z+=10;
}
 if( x>=4 && y <=2) error();
```

(a)

```
1  int i,j,k,n,m;
2
3  assume(n<=m);
4  for (i=0;i<n;i++)
5   for (j=0;j<n;j++)
6    for (k=j; k<n+m;k++)
7      assert(i+j<=n+k+m);
```

(b)

Figure 3.1: (a) Example from [33]. (b) Example `nested.c`.

Software model checking tools, e.g. [2, 47, 56], have previously used invariants from abstract interpretation—most notably alias analysis, but also octagonal constraints [56]—to strengthen the transition relation of the program. The contribution of this work to the research on software model checking is a powerful predicate inference engine using invariant generation. We also perform detailed comparisons of the benefits of combining invariant generation with abstract interpretation, as well as combining invariant generation with CEGAR-based software verification.

Pure dynamic analysis has been used to identify likely, but not necessarily correct, program invariants [24]. The technique uses program tests to evaluate candidate predicates from some a priori fixed database. The predicates that evaluate to true on all test runs are returned as likely invariants. The basic technique is not sound, as the test suite could be inadequate. Hence in a second step, the inferred invariants are provided to a verification-condition based program verifier. If the verifier succeeds, the combination of the dynamic step and the verification ensures program safety, while removing the need for providing manual invariants. However, there are some shortcomings of this technique. First, since the predicates are chosen from some fixed set (usually for efficiency in evaluation), the required program invariants may not fall into this fixed class. Second, the generated invariants are not in general inductive, therefore if the verifier fails, it is not evident if either a guessed invariant is wrong (that is, more tests should be generated to remove it from the discovered set), or if the guessed invariant does represent all reachable states, but is too weak to allow the verifier to complete the proof.

3.1 Example

We illustrate our idea using the example program `nested.c` shown in Figure 3.1(b). We want to construct an invariant that proves the assertion in line 7.

The core idea of our tool is to perform constraint-based invariant synthesis. Our algorithm automatically discovers, through an iterative process, that we need an invariant template to be a conjunction of four inequalities for each loop head. The invariants for intermediate locations (between loop heads) can be computed from assertions for these locations by propagating strongest postconditions (or weakest preconditions). For clarity of presentation, we shall only show details relevant to the first conjunct in each template. We use the template map η such that

$$\begin{aligned}\eta(4) = \quad & \alpha + \alpha_{\mathtt{i}}\mathtt{i} + \alpha_{\mathtt{j}}\mathtt{j} + \alpha_{\mathtt{k}}\mathtt{k} + \alpha_{\mathtt{m}}\mathtt{m} + \alpha_{\mathtt{n}}\mathtt{n} \leq 0 \wedge \\ & \cdots \wedge \cdots \wedge \ldots \ , \\ \eta(5) = \quad & \beta + \beta_{\mathtt{i}}\mathtt{i} + \beta_{\mathtt{j}}\mathtt{j} + \beta_{\mathtt{k}}\mathtt{k} + \beta_{\mathtt{m}}\mathtt{m} + \beta_{\mathtt{n}}\mathtt{n} \leq 0 \wedge \\ & \cdots \wedge \cdots \wedge \ldots \ , \\ \eta(6) = \quad & \gamma + \gamma_{\mathtt{i}}\mathtt{i} + \gamma_{\mathtt{j}}\mathtt{j} + \gamma_{\mathtt{k}}\mathtt{k} + \gamma_{\mathtt{m}}\mathtt{m} + \gamma_{\mathtt{n}}\mathtt{n} \leq 0 \wedge \\ & \cdots \wedge \cdots \wedge \ldots \ .\end{aligned}$$

To obtain an invariant map from these templates, we need to instantiate the set of parameters

$$\begin{aligned}\{\ & \alpha, \alpha_{\mathtt{i}}, \alpha_{\mathtt{j}}, \alpha_{\mathtt{k}}, \alpha_{\mathtt{m}}, \alpha_{\mathtt{n}}, \\ & \beta, \beta_{\mathtt{i}}, \beta_{\mathtt{j}}, \beta_{\mathtt{k}}, \beta_{\mathtt{m}}, \beta_{\mathtt{n}}, \\ & \gamma, \gamma_{\mathtt{i}}, \gamma_{\mathtt{j}}, \gamma_{\mathtt{k}}, \gamma_{\mathtt{m}}, \gamma_{\mathtt{n}}\ \}\ .\end{aligned}$$

We proceed by constructing a system of constraints, say Ψ, over the set of template parameters that imposes the invariant conditions on the template map, following a classical approach from the literature [16, 77]. We omit the details for brevity. Unfortunately, even for this small example, we obtain a system of non-linear arithmetic constraints which exceeds the capacity of our constraint solver. Our idea is to scale the invariant generation engine by using information obtained from abstract interpretation as well as from concrete and symbolic runs of the program.

We first observe that for this example, some components of the required invariants can be generated by techniques based on abstract interpretation, e.g., by using octagon and polyhedral domains [19, 67]. By running InterProc (using PPL) on this example, we obtain the following invariant map η_α that annotates the loop locations with valid assertions:

$$\begin{aligned}\eta_\alpha(4) = \ & \mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0\ , \\ \eta_\alpha(5) = \ & \mathtt{n} \geq \mathtt{j} \wedge \mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0 \wedge \mathtt{j} \geq 0 \wedge \mathtt{n} \geq 1\ , \\ \eta_\alpha(6) = \ & \mathtt{n} + \mathtt{m} \geq \mathtt{k} \wedge \mathtt{n} \geq \mathtt{j} + 1 \wedge \mathtt{n} \leq \mathtt{m} \wedge \\ & \mathtt{k} \geq \mathtt{j} \wedge \mathtt{i} \geq 0 \wedge \mathtt{j} \geq 0\ .\end{aligned}$$

While theoretically the analysis could have found all polyhedral relationships, in practice tools like InterProc employ several heuristics that sacrifice precision for speed. In this case, InterProc misses the inequality $\mathtt{n} + \mathtt{m} \geq \mathtt{i}$ valid at lines 5 and 6 and crucial for proving the assertion. Our algorithm takes the output generated by the abstract interpreter and uses it as an initial, static strengthening to support constraint based invariant generation.

In the second step, our algorithm collects dynamic information by executing the program. We first present a direct approach that uses program states to compute additional constraints that support invariant generation. Then, we show an extension that can handle unbounded collections of states. The extended method uses symbolic execution to collect such sets of states. We formalize these direct and symbolic approaches in Section 3.3.

Direct approach Our direct approach starts with a collection of some reachable program states, which can be obtained by applying test generation techniques. We only track states at the head locations of the loops. Suppose we get the following set of states $\{s_1, \ldots, s_4\}$ by running the program on test inputs:

$$\begin{aligned}s_1 = \ & (\mathtt{pc} = 4, \mathtt{i} = \mathtt{j} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1)\ , \\ s_2 = \ & (\mathtt{pc} = 4, \mathtt{j} = 3, \mathtt{i} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1)\ , \\ s_3 = \ & (\mathtt{pc} = 5, \mathtt{i} = \mathtt{j} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1)\ , \\ s_4 = \ & (\mathtt{pc} = 6, \mathtt{i} = \mathtt{j} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1)\ .\end{aligned}$$

Here, the variable pc represents the control location. We shall use these states to simplify the constraints for invariant generation.

We observe that since template expressions must be true for all reachable program states, in particular, they must hold for the states collected by testing. That is, for each reachable state we can substitute program variables appearing in the template by their values determined by the states and use this information to strengthen the constraint Ψ.

Thus, we can conjoin the following set of linear inequalities to the system of constraints Ψ, which determines the invariant map:

$$\begin{aligned} \alpha + \alpha_{\mathtt{m}} + \alpha_{\mathtt{n}} &\leq 0 , && \text{from } s_1 \\ \alpha + 3\alpha_{\mathtt{j}} + \alpha_{\mathtt{m}} + \alpha_{\mathtt{n}} &\leq 0 , && \text{from } s_2 \\ \beta + \beta_{\mathtt{m}} + \beta_{\mathtt{n}} &\leq 0 , && \text{from } s_3 \\ \gamma + \gamma_{\mathtt{m}} + \gamma_{\mathtt{n}} &\leq 0 . && \text{from } s_4 \end{aligned}$$

These additional constraints are linear. They can be applied by the solver to trigger a series of simplification steps. After the solving succeeds, we obtain the following invariant map:

$$\begin{aligned} \eta(4) &= \ \mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0 , \\ \eta(5) &= \ \mathtt{n} + \mathtt{m} \geq \mathtt{i} \wedge \mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0 , \\ \eta(6) &= \ \mathtt{n} + \mathtt{m} \geq \mathtt{i} \wedge \mathtt{k} \geq \mathtt{j} \wedge \mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0 . \end{aligned}$$

Symbolic approach We observe that we can simulate the effect of dynamic simplification using a large/unbounded set of reachable states. For this purpose we use symbolic execution, which computes assertions representing sets of reachable program states. We assume the example discussed so far and three reachable symbolic states below:

$$\begin{aligned} \varphi_1 &= (\mathtt{pc} = 4 \wedge \mathtt{i} = 0 \wedge \mathtt{n} \leq \mathtt{m}) , \\ \varphi_2 &= (\mathtt{pc} = 5 \wedge \mathtt{i} = 0 \wedge \mathtt{j} = 0 \wedge \mathtt{n} \geq 1 \wedge \mathtt{n} \leq \mathtt{m}) , \\ \varphi_3 &= (\mathtt{pc} = 6 \wedge \mathtt{i} = 0 \wedge \mathtt{j} = 0 \wedge \mathtt{k} = 0 \wedge \mathtt{n} \geq 1 \wedge \mathtt{n} \leq \mathtt{m}) . \end{aligned}$$

These symbolic states can be applied to derive additional linear constraints on the template parameters. Due to the reachability of φ_1, φ_2, and φ_3 the implications

$$\varphi_1 \rightarrow \eta(4) \ , \ \ \varphi_2 \rightarrow \eta(5) \ , \ \varphi_3 \rightarrow \eta(6)$$

hold for all valuations of program variables. The validity of these implications can be translated into a linear constraint, say Φ, over template parameters. (See Section 3.3 for details.) We conjoin the constraint Φ with the constraint Ψ that encodes the invariance condition. As a result, the solver performs additional simplifications that lead to improved running time.

Relevant strengthening In fact, after running our algorithm we can discover which inequalities computed using abstract interpretation and added as strengthening to the program were actually useful for finding the invariant that proves the assertion. This information is crucial for keeping minimal the number of facts reported to the software model checker as refinement predicates. For this purpose, we examine the solutions that the constraint solver assigned to the variables encoding the implication validity. For our example, the following inequalities found by InterProc were useful: $\mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0$ at line 4, $\mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0$ at line 5, and $\mathtt{k} \geq \mathtt{j} \wedge \mathtt{n} \leq \mathtt{m} \wedge \mathtt{i} \geq 0$ at line 6.

3.2 Constraint-based invariant generation

We start by describing the invariant-based approach for the verification of temporal safety properties and illustrate constraint-based invariant generation.

In the *constraint-based approach* [18,58,74,75,76] to invariant generation, the computation of an invariant map is reduced to a global constraint solving problem over the program locations. The approach consists of three steps. First, a *template* assertion that represents an invariant for each location is fixed in an *a priori* chosen language. A template assertion refers to the program variables V as well as a set of parameters. A parameter valuation determines an invariant. Second, a set of *constraints* over these parameters is defined in such a way that the constraints correspond to the definition of the invariant. This means that every solution to the constraint system yields a safe inductive invariant map. Third, a valuation of parameters is obtained by solving the resulting constraint system.

The language of arithmetic has been widely used to specify invariant templates [58, 74, 75]. A *linear inequality* over the variables $V = (x_1, \ldots, x_n)$ is an expression of the form $a_0 + a_1 x_1 + \ldots + a_n x_n \leq 0$ if $a_0, \ldots, a_n$ are rational numbers. The language of linear arithmetic consists of conjunctions of linear inequalities. An invariant template in linear arithmetic treats $\alpha_0, \ldots, \alpha_n$ as unknown parameters. For example, the template $\alpha + \alpha_x x + \alpha_y y + \alpha_z z \leq 0$ represents a linear inequality term over the variables x, y, and z. Here, the parameters are α, α_x, α_y, and α_z. A possible template instantiation is $-4 + x + 2y - z \leq 0$.

An invariant template and its expressiveness are determined by the number of conjuncts that appear in the template for each program location. Adding more conjuncts increases the expressive power at the cost of a more expensive constraint solving task. Usually, templates are constructed incrementally, by starting with the weakest template that assigns a single conjunct to each program location and then refining it by adding additional conjuncts if the constraint solving fails to instantiate the template.

Given a template specification for an invariant map, we generate a set of constraints that encode the inductiveness and safety conditions. To encode the inductiveness condition, we generate a constraint $\eta(\ell) \wedge \rho \rightarrow (\eta(\ell'))'$ for each transition (ℓ, ρ, ℓ'). Note that this implication is implicitly universally quantified over V and V'. Furthermore, the conjunction of such implications for all transitions is existentially quantified over the template parameters. Using Farkas' lemma [77], we eliminate universal quantification. The result is a set of existentially quantified non-linear constraints over the template parameters as well as over the parameters introduced by Farkas' lemma (see [74] for the technical details). Techniques involving Gröbner bases and real quantifier elimination can be used similarly to generate and solve constraints for more general polynomial constraints [58, 75], and for the combined theory of linear arithmetic and uninterpreted functions [4].

We assume a function InvGenSystem that computes constraints from programs and templates. An application of InvGenSystem on a program and templates for each program location produces a constraint over the template parameters that encodes the invariant map conditions. For the implementation details see [4, 16]. In this chapter, we present a constraint encoding that takes into account the assumption that only the error location can be unreachable. Thus, we only consider the case of the Farkas' lemma that deals with the implication between a *satisfiable* system of inequalities and an additional inequality.

We illustrate InvGenSystem using a single transition between location ℓ and ℓ' with the transition relation $x \leq y \wedge x' = x + 1 \wedge y' = y$. We assume a template $\varphi = (\alpha + \alpha_x x + \alpha_y y \leq 0 \wedge \beta + \beta_x x + \beta_y y \leq 0)$ consisting of two conjuncts at the location ℓ, and a singleton conjunction $\psi = (\gamma + \gamma_x x + \gamma_y y \leq 0)$ at the location ℓ'. The starting point is the implication $\varphi \wedge \rho \rightarrow \psi'$. To simplify the exposition, we first eliminate the primed program variables and obtain $\varphi \wedge x \leq y \rightarrow \psi[x+1/x]$, which we present in matrix form below.

$$\begin{pmatrix} \alpha_x & \alpha_y \\ \beta_x & \beta_y \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} -\alpha \\ -\beta \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} \gamma_x & \gamma_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq -\gamma - \gamma_x$$

Now, we apply Farkas' lemma to encode the validity of implication and obtain the following constraint:

$$\exists \lambda \geq 0.\ \lambda \begin{pmatrix} \alpha_x & \alpha_y \\ \beta_x & \beta_y \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} \gamma_x & \gamma_y \end{pmatrix} \wedge \lambda \begin{pmatrix} -\alpha \\ -\beta \\ 0 \end{pmatrix} \leq -\gamma - \gamma_x$$

This constraint determines the values of template parameters and the additional parameter λ. It contains non-linear terms that result from the multiplication of λ with $(\alpha_x\ \beta_x)$ and $(\alpha_y\ \beta_y)$.

Constraint Solving The constraints generated above are non-linear, since they contain multiplication terms over the parameters from the invariant templates, as well as the additional parameters introduced by

Farkas' lemma. The existing solving approaches include symbolic techniques based on instantiations and case splitting, e.g. [16], and using SAT solvers by applying an appropriate propositional encoding, e.g. [35]. Unfortunately, in all but the most basic programs, constraint-based invariant synthesis using the above technique is too expensive.

For the rest of the chapter, we assume a tool Solve that takes as input a set of non-linear constraints and returns either a satisfying assignment to the constraints, or that the constraint set is unsatisfiable, or times out. In this chapter we present techniques to increase the efficiency of solving by simplifying constraints that are passed to Solve.

3.3 Constraint simplification

We now describe how we can use additional static and dynamic information to restrict the search space determined by the set of static constraints. Technically, we do this by computing additional constraints on the program transition relation and on the template parameters and conjoining them with the constraint system defining invariant map. Program computations provide a source of such additional dynamic constraints.

InvGen+AbsInt: simplification from abstract interpretation

Our first simplification uses an abstract interpreter to compute program invariants, and uses the result of the abstract interpretation algorithm to strengthen the program transition relation. That is, suppose that η_α is an inductive invariant map computed by an abstract interpretation algorithm. In our constraint generation, we replace the constraint $\eta(\ell) \wedge \rho \rightarrow (\eta(\ell'))'$ for a transition (ℓ, ρ, ℓ') with the constraint $\eta(\ell) \wedge (\eta_\alpha(\ell) \wedge \rho) \rightarrow (\eta(\ell'))'$.

The following lemma formalizes the strengthening property of η_α.

Lemma 1. *For a given invariant template map η for the program $\mathcal{P}$, if an inductive invariant map η^* is obtained by strengthening of the transition relation of $\mathcal{P}$ with an inductive invariant map η_α then the map*

$$\lambda \ell \in \mathcal{L}.\ (\eta^*(\ell) \wedge \eta_\alpha(\ell))$$

is an inductive invariant map for $\mathcal{P}$.

Proof. The proof follows from the definition of inductive invariant maps. $\square$

InvGen+Test: simplification from tests

Individual program computations can be used to simplify the constraints for invariant generation. The crux of the algorithm InvGen+Test lies in the observation that an invariant template must hold when partially evaluated on a reachable state of the program.

Let $t(V)$ be a template over the program variables V and s be a reachable program state. We write $t(s/V)$ to denote a template expression that is obtained from t by substituting each variable $x \in V$ with its value $s(x)$ in the state s. Then, the constraint $t[s/V]$ imposes an additional constraint over the template parameters. Note that this constraint is *linear*, i.e., its processing does not require application of expensive non-linear solving techniques.

We show the algorithm InvGen+Test in Figure 3.2. The algorithm takes as input a program $\mathcal{P}$ and an invariant template map η with parameters $\mathcal{P}$. It can return an invariant map for $\mathcal{P}$, output that no invariant map exists for the given invariant templates, or find a counterexample to the program safety. There are three conceptual steps of the algorithm. The first step (line 1) constructs a set Ψ of constraints on the invariant template parameters that encode the initiation, inductiveness, and safety conditions. The second step (lines 2–9) runs a set of tests and generates additional constraints on the parameters based on the test executions. Finally, the third step (line 10) solves the conjunction of the static constraints from line 1 and the additional constraints generated during testing.

The loop in lines 3–9 executes the program on a set of tests. We instrument the program so that for each program location ℓ reached in the test, the concrete values of all the program variables that appear in the

input
$\mathcal{P}$: program;
η: invariant template map with parameters $\mathcal{P}$
vars
Ψ : static constraint;
Φ : dynamic constraint
begin
1 $\Psi := \mathsf{InvGenSystem}(\mathcal{P}, \eta)$
2 $\Phi := \mathit{true}$
3 **repeat**
4 $\langle \ell_1, s_1 \rangle, \dots, \langle \ell_n, s_n \rangle := \mathsf{GenerateAndRunTest}(\mathcal{P})$
5 **if** $\ell_n = \ell_{err}$ **then**
6 **return** "counterexample $\langle \ell_1, s_1 \rangle, \dots, \langle \ell_n, s_n \rangle$"
7 **else**
8 $\Phi := \Phi \wedge \bigwedge_{i=1}^{n} (\eta(\ell_i))[s_i/V]$
9 **until** no more tests
10 **if** $\mathcal{P}^* := \mathsf{Solve}(\Psi, \Phi)$ succeeds **then**
11 **return** "inductive invariant map $\eta[\mathcal{P}^*/\mathcal{P}]$"
12 **else**
13 **return** "no invariant map for template"
end.

Figure 3.2: Algorithm InvGen+Test for invariant generation supported by dynamic simplification using program executions. InvGenSystem creates a constraint over the template parameters that encodes invariant map conditions for the program $\mathcal{P}$, see Section 3.2. The function GenerateAndRunTest selects program computations.

template $\eta(\ell)$ are recorded. If a test hits the error location, then of course, we have found a bug, and we return this error (lines 5,6). Otherwise, the recorded values provide an additional constraint on the template parameters. For example, if the template for a location is $\alpha x + \beta y + \gamma \leq 0$, and a dynamic execution reaches this location with the concrete state $x = 35, y = -9$, we know that the parameters α, β, and γ must satisfy the constraint $35\alpha - 9\beta + \gamma \leq 0$. We call this a *dynamic constraint* on the parameters and add this constraint to the auxiliary constraint Φ.

The testing loop terminates due to an externally supplied coverage criterion. At this point, the constraint solver is invoked to find a satisfying assignment for the parameters in $\mathcal{P}$ that satisfy both the static constraints in Ψ and the dynamic constraints in Φ. If there is no such solution, the algorithm returns that there is no invariant map for the program using the current template map. On the other hand, any satisfying assignment provides an invariant map. Our algorithm maintains the invariant that at any point in lines 3–13, a satisfying assignment to the constraints $\Psi \wedge \Phi$ is guaranteed to be a valid invariant map.

The following lemma formalizes the strengthening from tests.

Lemma 2. *Algorithm* InvGen+Test *computes constraints Ψ and Φ such that Ψ implies Φ.*

Proof. By definition, Ψ constraints template parameters such that the resulting inductive invariant holds for every reachable state, i.e.,

$$\forall \mathcal{P}.\ \Psi \rightarrow \bigwedge \{\eta(\ell)[s/V] \mid s \text{ is reachable at } \ell\}\ .$$

Since Φ only considers a finite set of reachable states, it is implied by Ψ. □

InvGen+Symb: simplification from symbolic execution

We observe that the basic algorithm conjoins dynamic, *linear* constraints for each state that is reached by the test generator. A large number of such constraints may overwhelm the constraint solver, despite their

```
3      repeat
4.1       π := GeneratePath(P)
4.2       (* π = (ℓ1, ρ1, ℓ2), ..., (ℓn, ρn, ℓn+1) *)
5         if ℓn+1 = ℓerr and π is feasible then
6            return "counterexample π"
7         else
8.1          φ := ∃V''. (ρ1 ∘ ··· ∘ ρn[V''/V][V/V'])
8.2          Φ := Φ ∧ Encode(φ → η(ℓn+1))
9      until no more paths
```

Figure 3.3: Algorithm InvGen+Symb. It can be obtained by replacing lines 3–9 of the algorithm InvGen+Test with the above statements. The function GeneratePath selects program paths. Encode creates *linear* constraints over template parameters that encode the validity of the given implication.

low processing cost. We improve the basic algorithm by taking into account *sets* of reachable states using a single strengthening constraint.

We assume a template $t(V)$ and a set of reachable states represented by an assertion $\varphi(V)$. We can obtain such sets of states by performing symbolic execution along a collection of program paths. Then, the implication $\varphi(V) \to t(V)$ must hold for all valuations of V since every state in φ is reachable.

Following the method in Section 3.2, we encode the validity of the implication by a constraint over the template parameters. In this case, the encoding yields *linear* constraints. In contrast to the cases when the left-hand side of the implication contains template assertions, in the above implication program variables have *constant* coefficients. Thus, when multiplying additional parameters (appearing due to the application of Farkas' lemma) with coefficients attached to the program variables we obtain linear terms, which, in turn, result in linear constraints.

For example, we consider a template $t(x, y, z)$ that consists of two conjuncts $\alpha + \alpha_x x + \alpha_y y + \alpha_z z \le 0 \wedge \beta + \beta_x x + \beta_y y + \beta_z z \le 0$. We assume a set of states $\varphi = (-x \le 0 \wedge -y \le 0 \wedge x + y - z \le 0)$ reached by symbolic execution. The encoding of the implication $\varphi \to t$ yields the constraint

$$\exists \Lambda \ge 0.\ \Lambda \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & -1 \end{pmatrix} = \begin{pmatrix} \alpha_x & \alpha_y & \alpha_z \\ \beta_x & \beta_y & \beta_z \end{pmatrix} \wedge \Lambda \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \le \begin{pmatrix} -\alpha \\ -\beta \end{pmatrix} \ ,$$

which is clearly linear.

We assume a function Encode that translates an implication between an assertion representing a set of states and a template into a linear constraint over template parameters. Our extended algorithm InvGen+Symb applies Encode on sets of reachable states computed by symbolic execution of the program. The algorithm is presented in Figure 3.3. Since it extends the basic algorithm InvGen+Test by adding the symbolic treatment of reachable states, we only present the modified part.

The algorithm InvGen+Symb interleaves symbolic execution and collection of constraints. It relies on an external function GeneratePath that selects paths through the control flow graph of the program, see line 4.1. For a given path, we compute an assertion representing states that are reachable by executing its transitions, see line 8.1. We use the relational composition operator $\circ$, which is defined by $\rho \circ \rho' = \exists V''.\ \rho[V''/V'] \wedge \rho'[V''/V]$, to compute the transition relation of the whole path. The existential quantification in line 8.1 projects this relation to the successor states φ, i.e., it computes the range of the relation. We use variable renaming to keep the resulting assertion consistent with the templates over program variables. We conjoin the constraint resulting from the translation of the implication between the reachable states φ and the corresponding template $\eta(\ell_{n+1})$ to the dynamic constraint Φ before proceeding with the next path. We assume an external procedure that selects a finite set of paths. In our implementation, we apply directed symbolic execution that attempts to unroll loops at least one time.

The following lemma formalizes the strengthening from reachable symbolic states.

Lemma 3. *Algorithm* InvGen+Symb *computes constraints* Ψ *and* Φ *such that* Ψ *implies* Φ*.*

Proof. By definition, Ψ constraints template parameters such that the resulting inductive invariant holds for every reachable state, i.e.,

$$\forall \mathcal{P}.\ \Psi \rightarrow \bigwedge \{\eta(\ell)[s/V] \mid s \text{ is reachable at } \ell\} \ .$$

Since Φ only considers (a subset of) reachable states, it is implied by Ψ. $\square$

Correctness

We discuss the correctness of the algorithms InvGen+AbsInt, InvGen+Test, and InvGen+Symb. These algorithms are sound, i.e., they compute inductive invariant maps, since the constraint produces by *invGenConstraint* guarantees to restrict template parameters such that each satisfying valuation yields an inductive invariant map [16]. The soundness of InvGen+AbsInt also relies on the fact that the employed abstract interpretation tool computes inductive invariant maps.

The following theorem formalizes the completeness guarantees offered by InvGen+Test and InvGen+Symb. Lemma 1 discusses the completeness under strengthening from abstract interpretation.

Theorem 1. *[Correctness] Given program $\mathcal{P}$, if there is an inductive invariant map η^* that is an instantiation of the template map η then η^* satisfies the conjunction of constraints $\Psi \wedge \Phi$ computed by Algorithms* InvGen+Test *and* InvGen+Symb.

Proof. The proof follows from Theorem 2 in [16], which states the completeness of the constraint Ψ, together with Lemmas 2 and 3, which state that the strengthening is not eliminating any solutions of Ψ. $\square$

3.4 Template-guided coverage

We implement GenerateAndRunTest using both random test input generation and systematic test input generation using concolic execution [28, 29, 79]. We assume a location ℓ with a corresponding template $\alpha_0+\sum_{i=1}^{n} \alpha_i x_i \leq 0$. Our goal is to compute states at the location ℓ that produce as many linearly independent constraints on the parameters $\alpha_0, \ldots, \alpha_n$ as possible.

Simple location or branch coverage is inadequate, as it only guarantees one constraint on the template parameters. Discovery of too many states at the same program location can be not effective either. For example, consider a set of reachable valuations of program variables x and y, which is characteristic for loop unrolling sequences: $\{(1,2),(2,3),\ldots,(9,10)\}$. When executing the algorithm InvGen+Test, this set yields dynamic constraints

$$\begin{aligned} \alpha + \alpha_x \cdot 1 + \alpha_y \cdot 2 \leq 0 \ , \\ \alpha + \alpha_x \cdot 2 + \alpha_y \cdot 3 \leq 0 \ , \\ \cdots \ , \\ \alpha + \alpha_x \cdot 9 + \alpha_y \cdot 10 \leq 0 \ , \end{aligned}$$

which is equivalent to the conjunction

$$\alpha + \alpha_x + 2\alpha_y \leq 0 \wedge \frac{1}{9}\alpha + \alpha_x + \frac{10}{9}\alpha_y \leq 0 \ .$$

Thus, this example demonstrates that a large number of states does not necessarily provide a constraint that leads to a strong simplification (in a the sense of logical implication). Next, we present a coverage criterion that can be used for the selection of interesting paths through the control flow graph such that the resulting states deliver undiscovered simplifications, when available.

Given a location ℓ and a template $\alpha_0 + \sum_{i=1}^{n} \alpha_i x_i \leq 0$ we say that a set of states $S = \{s_1, \ldots, s_k\}$ forms a *basis* at ℓ for the template if 1) the size of the set is equal to the number of variables in the template, i.e., $k = n$, and 2) the states in S are linearly independent. The second condition is defined by the implication

$$\forall \lambda_1, \ldots, \lambda_k.\ \sum_{i=1}^{k} \lambda_i s_i = 0 \rightarrow \lambda_1 = \cdots = \lambda_k = 0 \ .$$

Then, the theoretical goal of full *basis coverage* is to generate a basis for each location and each template in the invariant template map. While the number of reachable states or paths of the program can be unbounded, the minimal number of tests to provide basis coverage is bounded by $|\mathcal{L}| \cdot |\eta| \cdot (|V| + 1)$, where $|\eta|$ gives the number of templates in the invariant template map.

Next, we show how a state-of-the-art test input generator based on symbolic execution, e.g. [28, 29, 79] can be extended to account for basis coverage. In such test input generator schemes, the program is executed on symbolic inputs, and a set of constraints on the symbolic inputs is maintained. The constraints encode the conditionals visited along the path. A satisfying assignment to the constraints guarantees a test input that produces a computation along the path taken by the symbolic execution.

Suppose that a location ℓ of the program has been visited by the states $s_1, \ldots, s_k$. We wish to find an additional state s that is linearly independent of the states $s_1, \ldots, s_k$. We accomplish this task by providing an additional constraint to the path constraint collected by the symbolic execution. This additional constraint gives the most general condition for an state s to be linearly independent of $s_1, \ldots, s_k$. It is generated using elementary linear algebra.

First, we find a state v in the orthogonal complement of the subspace spanned by $s_1, \ldots, s_k$. This is any non-zero state that is orthogonal to each of states in the set $s_1, \ldots, s_k$. We encode this condition by the following constraint:

$$v \cdot s_1 = 0 \wedge \cdots \wedge v \cdot s_k = 0 \wedge v \neq 0 \ ,$$

where $\cdot$ represents vector dot product, and the dimension of each vector is n. The constraints with product terms enforce that v is orthogonal to each vector s_i. The last conjunct enforces v to be non-zero, which is equivalent to

$$v_1 \neq 0 \vee \cdots \vee v_n \neq 0 \ .$$

These constraints form a linear system and can be solved using Gaussian elimination.

Given a vector v that satisfies the constraints above, we add the constraint that the new state s is not orthogonal to v. That is, we require that the new state has a non-zero projection on the vector v. This is formalized by the constraint

$$s \cdot v \neq 0 \ ,$$

which is a linear constraint, since v is constant and the only unknown values are the components of s. This guarantees that the new state is linearly independent of all the previous ones.

Example 1. [Basis coverage] We illustrate the computation of additional states increasing the basis coverage for the set of states $\{s_1, s_2\}$ over the variables x, y, and z such that $s_1 = (1, 2, 0)$ and $s_2 = (0, 1, 2)$.

We constrain the orthogonal state $v = (v_1, v_2, v_3)$:

$$v_1 + 2v_2 = 0 \wedge v_2 + 2v_3 = 0 \wedge (v_1 \neq 0 \vee v_2 \neq 0 \vee v_3 \neq 0) \ ,$$

which has a satisfying assignment $v = (1, -\frac{1}{2}, \frac{1}{4})$. Then, we find the additional state s by solving the constraint

$$s_1 - \frac{1}{2}s_2 + \frac{1}{4}s_3 \neq 0 \ .$$

We obtain the state $s = (0, -2, 1)$.

Assume that the path relation leading to the location under consideration is

$$\rho = \ x \geq 0 \wedge x' = x - 2 \wedge y' = y \wedge z' = z + 1 \ .$$

Then, solving the conjunction of the above constraints together with $\rho[s/V]$ yields an initial state $(2, -2, 0)$ that leads to the additional state that increases the basis coverage.

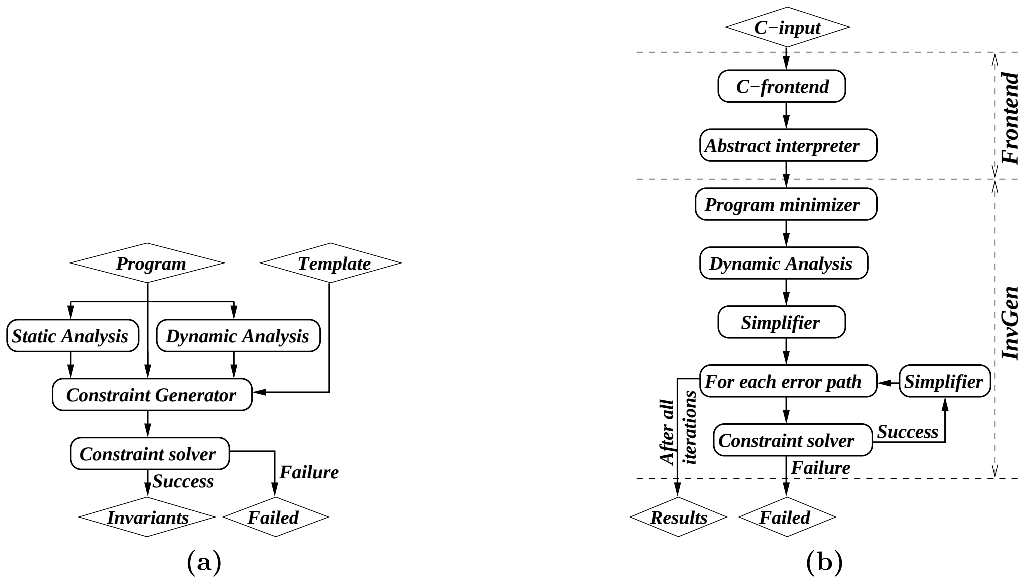


Figure 3.4: (a) InvGen design. (b) InvGen implementation.

3.5 InvGen: invariant generator

Design

We present the design of InvGen in Figure 3.4(a). The input program is passed to the dynamic and static analyzers. The results of each analysis together with the program and the templates are passed to the constraint generator. The generated constraints are solved by a constraint solver. If the solver succeeds then InvGen returns a safe invariant.

Implementation

Figure 3.4(b) outlines the implementation of InvGen. It is divided into two executables, *frontend* and InvGen. The frontend executable contains a CIL [69] based interface to C and an abstract interpreter InterProc [60]. The frontend takes a program procedure written in C language as an input, and applies InterProc on the program three times using the interval, octagon, and polyhedral abstract domains. Then, the frontend outputs the transition relation of the program that is a annotated with the results computed by InterProc. See [41] for the output format.

Next, we describe the components of InvGen, following Figure 3.4(b).

Program minimizer InvGen minimizes the transition relation of the program to reduce the complexity of constraint solving. InvGen computes a minimal set of cut-point locations, and replaces each cut-point free path by a single, compound program transition. The unsatisfiable and redundant transitions are eliminated. At this phase, the invariants obtained from InterProc can lead to the elimination of additional transitions.

Dynamic analysis InvGen collects dynamic information for the minimized program using concrete and symbolic execution. In case of concrete execution, InvGen collects a finite set of reachable states by using a guided testing technique. Otherwise, InvGen performs a bounded, exhaustive symbolic execution of the program. By default, the bound is set to the number of cut-points in the program. The user can limit the maximum number of visits for each cut-point during the symbolic execution.

File	InterProc	InvGen	InvGen+Z3	InvGen + InterProc	InvGen+Z3 + InterProc +Symb	InvGen + InterProc +Symb
Seq	×	23.0s	1s	**0.5s**	6s	**0.5s**
Seq-z3	×	23.0s	9s	**0.5s**	6s	**0.5s**
Seq-len	×	T/O	T/O	T/O	4s	**2.8s**
nested	×	T/O	T/O	17.0s	3s	**2.3s**
svd(light)	×	T/O	T/O	**10.6s**	T/O	14.2s
heapsort	×	T/O	T/O	19.2s	48s	**13.3s**
mergesort	×	T/O	**52s**	142s	T/O	170s
SpamAssassin-loop	✓	T/O	5s	**0.28s**	1s	0.4s
apache-get-tag	×	**0.4s**	10s	0.6s	3s	0.7s
sendmail-fromqp	×	0.3s	5s	0.3s	5s	**0.3s**
Example1(b)	×	T/O	T/O	0.4s	1s	0.35s

Table 3.2: Comparison of variations of invariant verification techniques and InterProc on additional benchmark problems inspired by [59]. "✓" and "×" indicate whether the invariant computed by InterProc proves the assertions, and "T/O" stands for time out.

Simplifier InvGen simplifies all arithmetic constraints locally at each step of the algorithm. InvGen also simplifies constraints obtained by the concrete execution and abstract interpretation.

Constraint solver The inductiveness conditions result in non-linear arithmetic constraints. In practice, the existentially quantified variables range over a small domain, typically they are either 0 or 1. InvGen leverages this observation in order to solve the constraints by performing a case analysis on the variables with small domain. Each instance of case analysis results in a linear constraint over template parameters, which can be solved using a linear constraint solver. This approach is incomplete, since InvGen does not take all possible values during the case analysis, however it is effective in practice.

Multiple paths to error location A program may have multiple cut-point free paths that lead to the error location, which we refer to as error paths. InvGen deals with multiple error paths in an incremental fashion for efficiency reasons. Instead taking inductiveness conditions for all error paths into account, InvGen computes a safe invariant for one error path at a time. Already computed invariants are used as strengthening when dealing with the remaining error paths.

3.6 Experiments

Implementation We implemented the algorithms InvGen+Test and InvGen+Symb using SICStus Prolog [82], the linear arithmetic solver clp(q,r) [52] and the Z3 solver [21] as the backend to solve non-linear constraints. When describing the application of InvGen together with Z3, we shall write InvGen+Z3. We apply the InterProc [60] tool for abstract interpretation over numeric domains, and use the PPL backend for polyhedra, mainly due to its source code availability. In principle, a variety of other tools could be used instead, e.g., the ASPIC tool implementing the lookahead widening and acceleration techniques [31, 32]. InvGen provides a frontend for C programs, which relies on the CIL infrastructure for C program analysis and transformation and abstracts from non-arithmetic operations appearing in the input program. We

File	Blast	Blast + InvGen + InterProc + Symb
Seq	diverge	8
Seq-len	diverge	9
fregtest	diverge	3
sendmail-fromqp	diverge	10
svd(light)	144	43
Spamassassin-loop	51	24
apache-escape	26	20
apache-get-tag	23	15
sendmail-close-angle	19	15
sendmail-7to8	16	13

Table 3.3: InvGen + InterProc + Symb for predicate discovery in Blast. We show the number of refinement steps required to prove the property.

implement the following additional variable elimination optimization. The additional constraints obtained from dynamic and static strengthening are linear. In particular, the additional variables that encode implication between symbolic states and templates, Λ in the previous section, can be eliminated. We perform this simplification step before applying the (expensive) techniques for solving non-linear constraints. For our constraint logic programming-based implementation, this results in a reduction of the number of calls to the linear arithmetic solver. When using the SAT approach, it allows us to avoid applying the propositional search to constraints that can be solved symbolically.

In our experimental evaluation, we observed that InvGen+Test and InvGen+Symb offer similar efficiency improvement, with a few exceptions when InvGen+Symb was significantly better. To keep the tables with experimental data compact, we only describe evaluation of the strengthening that uses symbolic execution InvGen+Symb.

Software Verification Challenge Benchmarks We applied InvGen on a suite of software verification challenge programs described in [59]. The examples in this benchmark are extracted from large applications by mining a security vulnerability database for buffer overflow problems. We use the corrected versions of these programs, using the buffer access checks as assertions. The suite consists of 12 programs.[1] Using a polyhedral abstract domain, InterProc computes invariants that are strong enough to prove the assertion for half of them. The constraint based invariant generation together with the SAT-based encoding, i.e., InvGen+Z3, generates invariants for all programs within 36.5 seconds of total time. Using the CLP backend, InvGen handles 11 examples within 6.3 seconds, and times out on one program, which is handled by InvGen+Z3 in 5 seconds. Using the static and dynamic strengthening described in this chapter, we obtain the following running times. The combination InvGen+Z3+InterProc+Symb solves all examples in 29.5 seconds, while InvGen+InterProc+Symb handles all examples within 9.6 seconds. These experiments demonstrate that the various optimizations can have an effect on verification, but the running times were too short to draw meaningful conclusions.

Impact of Dynamic Strengthening The collection from [59] did not allow us to perform a detailed benchmarking of our algorithm, since the running times on these examples were too short. We obtained a set of more difficult benchmarks inspired by [59] by adding additional loops and branching statements, and provide a detailed comparison that describes the impact of static and dynamic strengthening in isolation in Table 3.2. InterProc computes 50 inequalities for each loop head, which results in a significant increase in the number of variables in the constraint system. While being an obstacle for the propositional search procedure in Z3, the increased number of variables does not significantly affect the CLP-based backend since

[1] Due to short running times, we present the aggregated data and do not provide any table containing entries for individual programs.

the additional variables appear in linear terms. In summary, the performance of InvGen+Z3 decreases and the performance of InvGen goes up by adding facts from InterProc.

Integration with Blast We have modified the abstraction refinement procedure of the Blast software model checker [46] by adding predicate discovery using path invariants [5]. Table 3.3 shows how constraint based invariant generation can be effective for refining abstractions. The number of counterexample refinement iterations required is reduced in all examples. For several examples we achieved termination of previously diverging abstraction refinement, and for others the reduction ranges between 25 and 400 percent.

Summary Our experimental evaluation leads to the following observations:

- For complex constraint solving problems, the additional strengthening facilitates significant improvement. It ranges from reducing the running time by two orders of magnitude to making timing out examples solvable within seconds.
- If the constraint solving is already fast in the purely static case, then the strengthening does not cause any significant running time penalty.

Chapter 4

Bound synthesis

During an execution, a program may consume various kinds of resources such as time, memory, or network bandwidth. Excessive resource consumption may affect the usability of the program. Static bounds on resource consumption can be a useful source of information when ensuring the program usability.

In this chapter, we present a constraint-based method for computing symbolic bounds on consumption of a resource. Our method takes a parameterized program as input. A parameterized program contains a set of *parameters* that are read only variables and a *consumption variable* that represents consumption of the resource. The parameters get a constant value during execution. We also assume that the parameterized program is instrumented with updates on the consumption variable such that value of the consumption variable represents consumption of the resource at each program step. Our method returns a linear expression over parameters that is an upper bound over the consumption variable in all possible executions of the program.

Our method is a modification of the template based invariant generation presented in Chapter 3. For each program location, we aim to compute an invariant and a upper bound on the consumption variable such that the invariant implies the upper bound. Along with a template for invariant at each program location, our method also assumes a template that expresses a space of linear upper bounds over parameters. Using these templates, we generate constraints and solve them as we presented in Chapter 3. A solution of the templates provides the symbolic expressions that bounds consumption variable. The generated constraints are very hard to solve. To gain efficiency in solving, we consider different paths in the program separately. We compute bounds for each path using the above method. We combine bounds for different paths and produce bound for the entire program.

In the next chapter, we apply the above method to develop a tool for C-to-gates hardware synthesis. C programs are widely written with the use of dynamically allocated memory. Dynamically allocated and manipulated data structures cannot be translated into hardware unless there is a constant upper bound on the amount of memory that the program uses during all executions. This bound can depend on the *parameters* to the program, *i.e.*, program inputs that are instantiated at hardware synthesis time. We use the constraint based method for the discovery of memory usage bounds, which leads to the first-known C-to-gates hardware synthesis supporting programs with non-trivial use of dynamically allocated memory, *e.g.*, linked lists maintained with `malloc` and `free`. We illustrate the practicality of our tool on a range of examples.

4.1 Resource bound analysis

Preliminaries For a program $\mathcal{P} = (V, \mathcal{L}, \ell_{init}, \mathcal{T}, \ell_{err})$, parameters $S \subset V$, and a resource consumption variable $h \in V \setminus S$, a parameterized program $\hat{\mathcal{P}} = (\mathcal{P}, S, h)$. Each transition relation preserves the values of parameters, *i.e.*, for each $(\ell, \rho, \ell') \in \mathcal{T}$ we have

$$\forall V\ \forall V' : \rho \rightarrow S' = S\ .$$

```
procedure BoundGen
input
  P̂ = ((V, L, ℓ_init, T, ℓ_err), S, h): parameterized program
  η^T: invariant template map with η^T(ℓ_init) = true
  Bnd^T: bound template map with Bnd^T(ℓ_init) = h ≤ 0
vars
  Q: template parameters in η^T and Bnd^T
  Ψ: auxiliary constraint over Q
begin
1   Ψ := true
2   for each ℓ ∈ L do
3      Ψ := Ψ ∧ ∀V : η^T(ℓ) → Bnd^T(ℓ)
4   for each (ℓ, ρ, ℓ') ∈ T do
5      Ψ := Ψ ∧ ∀V ∀V' : (η^T(ℓ) ∧ ρ) → η^T(ℓ')'
6   Q := free variables in Ψ
7   if exists M such that Ψ[M/Q] then
8      return Bnd^T[M/Q]
9   else
10     throw "no bound found"
end
```

Figure 4.1: BoundGen discovers bounds on the value of the variable h, which keeps track of the amount of consumption of a resource.

We are interested in a parametric invariant map *Bnd* that bounds the consumption variable. Formally, we will search for *Bnd* such that for each $\ell \in \mathcal{L}$ we have

$$\forall S \; \exists c \in \mathbb{N} \; \forall V \setminus S : \mathit{Bnd}(\ell) \rightarrow h \leq c \ .$$

For given values of parameters in S, the minimum constant c that satisfies above equation for all program locations determines the maximal amount of resource used during the program execution. For proving that *Bnd* is valid we will need an inductive invariant map η. Formally, we require that for each $\ell \in \mathcal{L}$ the following holds:

$$\forall V : \eta(\ell) \rightarrow \mathit{Bnd}(\ell) \ .$$

Bounds analysis algorithm Fig. 4.1 presents our constraint-based procedure BoundGen for discovering bounds on consumption variable. The procedure takes as parameters a parameterized program $\hat{\mathcal{P}}$, an invariant template map η^T, and a bound template map Bnd^T. It returns either a valid bound map or an exception if no such map can be found.

The template maps used by BoundGen are reminiscent of those used in in Section 3.2. The bound template map Bnd^T given to BoundGen as input assigns to each program location a bound template of the form

$$h \leq \delta_1 p_1 + \cdots + \delta_m p_m + \delta \ ,$$

where $\delta_1, \ldots, \delta_m, \delta$ are *template* parameters and $S = \{p_1, \ldots, p_m\}$ are parameters of $\hat{\mathcal{P}}$. Since Bnd^T only refers to S and h, it guarantees to yield parametric bound invariants only.

BoundGen collects a conjunction of constraints Ψ over template parameters for both template maps in lines 1–5. These constraints encode the condition that the computed bounds must be valid. Lines 2–3 state that the bounds hold for all reachable states, which are represented by an invariant map induced by the invariant template map η^T. Lines 4–5 encode the condition that η^T in fact represents all reachable program states.

procedure PATHBOUND
input
$\hat{\mathcal{P}} = ((V, \mathcal{L}, \ell_{init}, \mathcal{T}, \ell_{err}), S, h)$: parameterized program
η^T: invariant template map
Bnd^T: bound template map
var
Bnd : bound map
ℓ_{err}: distinguished error location
$\mathcal{T}_{\mathcal{E}}$: transitions for bound assertion checking
function PATHPROGRAM
input
π : sequence of transitions
begin
1 **return** $(V, \mathcal{L}, \ell_{init}, \{\tau \mid \tau = (\ell, \rho, \ell')$ occurs in π and $\ell' \neq \ell_{err}\}, \ell_{err})$
end;
begin
2 $Bnd := \lambda \ell \in \mathcal{L}.h \leq 0$
3 **repeat**
4 $\mathcal{T}_{\mathcal{E}} := \{(\ell, \neg Bnd(\ell) \wedge V' = V, \ell_{err}) \mid \ell \in \mathcal{L}\}$
5 **if** exists $\pi \in (\mathcal{T} \cup \mathcal{T}_{\mathcal{E}})^*$ from ℓ_{init} to ℓ_{err} such that $\rho_\pi \neq \emptyset$ **then**
6 $\mathcal{P}_\pi :=$ PATHPROGRAM(π)
7 **try**
8 $Bnd_\pi :=$ BOUNDGEN$((\mathcal{P}_\pi, S, h), \eta^T, Bnd^T)$
9 **catch**
10 **return** "unbounded consumption path π"
11 $Bnd := \lambda \ell \in \mathcal{L}.Bnd(\ell) \vee Bnd_\pi(\ell)$
12 **else**
13 **return** "bound assertion map Bnd"
14 **done**
end

Figure 4.2: PATHBOUND performs an incremental boundedness analysis using guidance from spurious counterexamples.

We collect all template parameters in line 6. If our constraint solving procedure can find a satisfying assignment to Ψ, then this assignment defines a bound map in line 8. Otherwise, BOUNDGEN raises an exception.

The transition relations in the program $\mathcal{P}$ produced during the shape analysis phase are conjunctions of linear inequalities over V and V'. For our templates consisting of linear inequalities, we eliminate the universally quantification over V and V' in lines 3 and 5 of BOUNDGEN by applying a standard technique, see *e.g.* [16], based on Farkas' lemma [26]. The resulting constraint Ψ is a conjunction of non-linear inequalities and can be efficiently solved using INVGEN. We implemented our algorithm in the ARMC model checker [71].

The soundness and completeness of BOUNDGEN is formalized in the following theorem.

Theorem 2. *The procedure* BOUNDGEN *is complete for bound expressions in linear arithmetic provable using linear arithmetic invariants,* i.e.*, in this case it computes a bound map. The procedure* BOUNDGEN *is also sound,* i.e.*, it computes a bound map that represents an upper bound on the resource consumption.*

Proof. We rely on the soundenes and completeness of the translation of the bounds synthesis problem to constraint solving. The translation follows the classical scheme applied for the synthesis of inductive invariants using constraint solving. □

Path bounds analysis The constraint-based procedure BoundGen performs an expensive computation—non-linear constraint solving—and does not scale beyond medium-sized programs. We improve the scalability of BoundGen by performing the boundedness analysis in an incremental fashion using the idea of path invariants [5]. We apply the expensive, constraint-based procedure only to certain program fragments, which are determined automatically.

Fig. 4.2 presents our BoundGen-based procedure PathBound for an incremental discovery of bounds on consumption variable for the entire program from its fragments. Initially, the bound map states that no consumption variable is zero, see line 2. Then, this claim is verified in line 5 using a verification tool for proving program safety. Such a tool is applied on an augmented program that is obtained from $\hat{\mathcal{P}}$ by adding a distinguished error location ℓ_{err} that is reachable if the consumption bound claimed by *Bnd* is not valid. In the case of a false bound, the algorithm will return a counterexample in the form of a sequence of transitions π that leads to consumption beyond the claimed bound.

In case a counterexample π is found, we identify a fragment of $\hat{\mathcal{P}}$ that is traversed by the transitions occurring in π. This code fragment is defined by a path program $\mathcal{P}_\pi$ for π [5], see line 1. In particular, the path program $\mathcal{P}_\pi$ contains the same loops of $\hat{\mathcal{P}}$ that are visited by π.

We compute an adjustment Bnd_π for the bound map by applying the procedure BoundGen on the path program, see line 8. The adjustment is used to weaken the claimed bound, see line 11.

This sequence of incremental adjustments continues until either the full program $\hat{\mathcal{P}}$ satisfies the claimed bound map or a path that for which no bound on consumption can be found is discovered.

The soundness and completeness properties of PathBound are inherited from the procedure BoundGen and the notion of path invariants.

Theorem 3. *The procedure* PathBound *is complete for bound expressions in linear arithmetic provable using linear arithmetic invariants,* i.e.*, in this case it computes a bound map and terminates. The procedure* PathBound *is also sound,* i.e.*, it computes a bound map that represents an upper bound on the resource consumption.*

Proof. We rely on the fact that the computed path programs grow by at least one transition at each iteration. Once all program transitions appear in the path program, Theorem 2 applies. $\square$

Chapter 5

C-to-gates synthesis using BoundGen

C-to-gates synthesis promises to bring the power of hardware based acceleration to mainstream programmers and to radically increase the productivity of digital designers [42]. However, today's C-to-gates synthesis tools do not support one of the most powerful and widely used features of high-level programming in C—dynamically allocated data structures. This leads to the use of arrays and significantly more complicated code for modelling naturally dynamic data structures with static data structures, which in turns incurs extra cost due to the extra complexity of design, verification, and maintenance. The support for dynamic memory abstraction remains an on-going research problem because of the need to efficiently and accurately determine a bound on heap consumption.

This work advances the state-of-the-art in hardware synthesis by providing support for programs that dynamically allocate, deallocate, and manipulate heap-based data structures. First, our approach for finding symbolic bounds uses shape analysis (*e.g.* [22, 61, 63]) and abstraction methods based on the introduction of new variables (*e.g.* [55, 62]) to produce a numerical program. Then, we use the above constraint-based method for finding a symbolic bound on the maximum heap size at compile time. This symbolic bound is expressed as a linear function on the *generic parameters* to the circuit description. The term generic parameter is used in hardware design languages to describe variables whose values will be known at compile-time. These generic parameters are simply referred as parameters in our method. With our method for computing symbolic bounds we can then automatically translate C programs with dynamic memory usage into equivalent programs that operate over statically allocated arrays. That is, when circuit descriptions are instantiated in their surrounding designs, the symbolic bounds can be used to compute concrete bounds for use during hardware synthesis.

Our method increases the expressive power available to the users of synthesis systems. For example, with our new C-to-gates synthesis flow, a designer can think in terms of a tree-based data structure, yet generate hardware that operates on a flat fixed sized array. Furthermore, off-the-shelf libraries can now be used as subroutines by digital designers. This leads to better re-use, as well as new avenues of adapting software verification techniques for use in hardware systems.

Our experiments show that it is possible to produce viable circuits from C programs that use dynamic data structures. By viable we mean that the synthesized circuits have performance that is good enough so that we see a possibility to significantly improve it with future work. This claim needs empirical justification by producing and analyzing the hand-coded equivalents. However, the generated circuits have a size and operating frequency which seems quite plausible.

Related work C-to-gates synthesis is a maturing field with notable systems—see [10,12,30,43,54,68,80,84]. Some existing C-to-gates synthesis systems already support pointers and pointer aliasing, see *e.g.* [78], but they do not deal with dynamically allocated data structures.

Synthesis tools for other general purpose programming languages also exist (*e.g.* tools supporting Scheme [73], or Haskell [6]). In a few rare instances (*e.g.* [9]) tools have been used not only to generate hardware but also the circuit's correctness proof as well. These tools usually require the user to estimate the maximal amount of memory allocated by the program and take this quantity as an input parameter to the synthesis routine. Thus, the results of our work can perhaps be used with these existing tools.

In the domain of pure functional programming languages, the topic of heap-bounds analysis has been extensively investigated, see *e.g.* [49]. For imperative programs, [50] develops a type system which tracks memory consumption. The Java memory-bounds tool described in [1] uses a heap abstraction and applies heuristics based on arithmetic simplification to find a memory bound. In contrast, our method uses a more precise numerical abstraction for dealing with heap, as we keep track of the size of intermediate list segments identified by the shape analysis when dissecting the heap, which was crucial for dealing with our examples. Furthermore, instead of using heuristics for finding the bound expression, we apply a constraint based boundedness analysis which is complete for linear bound expressions provable using linear invariants.

The semi-manual technique proposed in [8] uses Daikon [25] to collect likely program invariants—including facts about memory consumption—and uses them to derive an initial set of bound candidates.

In principle, the existing techniques for proving computational complexity, *e.g.* [34], can be used as a basis to design an algorithm for discovery of memory usage bounds. However, since we are only interested in bounds expressed over generic parameters, a major challenge is to bias the bound discovery method towards such well-formed bounds. Our constraint based procedure solves this challenge.

5.1 From heaps to arrays

In this section we describe an analysis that automatically discovers symbolic bounds on the heap usage. We will assume that the size parameters passed to `malloc` are fixed constants. Through the use of static analysis, we annotate each call to `free` with the amount of memory the call is freeing. For example, we would transform the call `free(tmp)` from Fig. 5.1 to `free(tmp,sizeof(LINK))`. For simplicity of presentation we will assume that programs allocate and free heap cells of a single fixed size. We can support multiple size allocations through the use of compile-time partial evaluation, but at the cost of complexity in the notation in this section. We currently do not support arbitrary DAGs or hash-tables, due to the limitations of existing separation logic based shape analysis tools [13, 22, 61, 63] of which we are dependent.

Our procedure is divided into the following steps.

Numerical heap abstraction First, we augment the program with a new variable h, which is used to track the amount of heap that is currently allocated. The variable h is incremented when malloc is called, and decremented when free is called. For memory-safe programs such behavior of h is correct. We use the shape analysis tool THOR [63] to determine the shape of the data structures used during the program's execution, and to prove memory safety. Using techniques from [62], THOR can be used to produce a new program without heap that is a sound abstraction of the original program—additional integer variables are added by THOR to summarize the sizes of data-structures. Thus, bounds found on h in the abstraction imply bounds in the original program. Note that the new program variables range over integers of arbitrary size (*i.e.* they cannot be represented in 32 or 64 bits).

The new abstract program is used for computing bounds on heap consumption only, and does not play any role during the hardware synthesis step.

Numerical bounds analysis Next, we apply our constraint-based boundedness analysis to the numeric program to find a symbolic bound f on the maximum value of h. For improved scalability we combine our constraint-based synthesis approach with a counterexample-guided method of checking and refining candidate bounds as presented in Section 4.1.

Array-based heap management and synthesis Once we have computed a symbolic bound (assuming that a bound can be found) we throw away the abstraction and then convert the original program into an array-based program operating over a pre-allocated shared array and then apply off-the-shelf synthesis tools to produce a gate-level design. Note that, although we may sometimes compute a conservative over-approximation for a bound on memory usage, it is often the case that a downstream synthesis tool can

```
void prio(int n,in_signal i,out_signal o) {
  LINK *tmp,*c,*buffer;
  assert( n>0 );
  while (1) {
    buffer = NULL;
    //Build up an n-sized sorted buffer
    for (int k=0;k<n;k++) {
     buffer = sorted_insert(input(i),buffer);
    }
    //Send the sorted list to the output and
    //deallocate the buffer as we walk it
    c=buffer;
    while(c!=NULL) {
      output(o,c->data);
      tmp = c;
      c = c->next;
      free(tmp);
    }
  }
}
```

(a)

```
LINK * sorted_insert(int data, LINK *l){
  LINK * elem = l;
  LINE * prev = NULL;
  LINK * x = (LINK*)malloc(sizeof(LINK));
  assert(x!=NULL);
  x->data = data;
  while (elem != NULL){
    if (elem->data >= x->data){
      x->next = elem;
      if (prev == NULL){l = x; return l;}
      prev->next = x;
      return l;
    }
    prev = elem;
    elem = elem->next;
  }
  x->next = elem;
  if (prev == NULL){l = x; return l;}
  prev->next = x;
  return l;
}
```

(b)

Figure 5.1: (a) Priority queue circuit specification in C, using off-the-shelf implementation of `sorted_insert`. The *generic* parameter `n` is assumed to be specified at compile-time. (b) Off-the-shelf implementation of incremental insertion sort procedure.

perform further pruning to yield a gate level implementation that does indeed have a better (or even ideal) bound. A simple case of this scenario is when a list is used to represent a bit-vector which is used in arithmetic expressions with known range at synthesis time allowing some of the upper bits to be pruned.

5.2 Example

Imagine that we would like to build an `n`-size priority queue circuit that reads integers from an input signal and returns every `n` input integers on an output signal in sorted order. See the function `prio` in Fig. 5.1(a) for an example of how we might wish to write a specification of the desired hardware in C. Our intention is that the variable `n` in Fig. 5.1(a) is a parameter, whereas `i` and `o` should be thought of as signal names. Our synthesis tool treats these in a special way as standard C, of course, does not make this distinction. In this example we assume that the circuit uses `input()` and `output()` as primitives for I/O on the signal variables `i` and `o`. `LINK` is a C struct used to represent singly-linked lists (with fields `data` and `next`). We make use of an existing off-the-shelf insertion-sort implementation, `sorted_insert`. See Fig. 5.1(b) for the source code of `sorted_insert`.

Note that in order to convert this program into hardware we must first find an *a priori* bound on the amount of heap during the execution of `prio`, for any input or parameter. The problem is that `sorted_insert` does not guarantee a concrete bound on the amount of heap allocated during its execution, instead it preserves a bound – it takes a state where k heap cells have been allocated and returns a state in which $k+1$ have been allocated. Thus we must hope to find a bound on the amount of heap used by `sorted_insert` from states limited to those reachable from `prio`.

If we can find this bound, then we can convert the program's operations on the heap into operations on statically-allocated arrays, thus facilitating synthesis. We aim to find a bound that holds across the entire program, but is expressed symbolically using only the generic parameters to the top-level function (*i.e.* the parameter `n` of the circuit `prio`). This allows us to pre-allocate a shared array when creating instances of

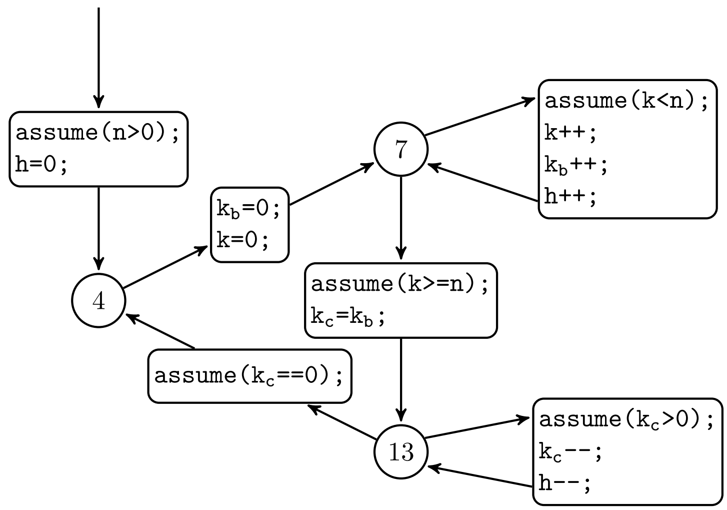


Figure 5.2: Numerical abstraction of procedure `prio` shown from Fig. 5.1(a).

the circuit `prio`.

The method given in Section 5.1 is designed to find a function f such that it is a program invariant that $f(\mathtt{n})$ is larger than the number of heap cells allocated at any given time during its execution. In this case the procedure described later will find the function $f(\mathtt{n}) = \mathtt{n} * 8$, assuming that `sizeof(LINK)` $= 8$ in the encoding.

With f we can now re-encode the program using a pre-allocated array. In essence, when we know the valuations to the input parameters we can then pre-allocate an array using f. We then convert dereferences like `*c` into `a[c]`. Field offsets are explicitly encoded: `c->data` is encoded as `a[c+0]`, and `c->next` is encoded as `a[c+4]`.

From this program (and via a translation into VHDL) we then used the Altera Quartus II 9.0 tools to construct an implementation for the Stratix III FPGA architecture. Using default synthesis and implementation options and with $\mathtt{n} = 10$, the generated circuit uses 5859 adaptive look-up tables, 4598 logic registers and 8192 block memory.

The following subsections apply the three steps of our method on the example and describe our method in detail.

Numerical heap abstraction

A shape analysis tool is designed to take a program and compute an invariant for each program location describing the shape of the heap. The invariant describes the data structures stored in the heap during the program's execution. Shape analysis tools are based on symbolic simulation together with abstraction techniques.

Using techniques described in [62], the shape analysis tool Thor can be used to introduce new variables which soundly track the sizes of data structure shapes inferred by the shape analysis. In the example of the function `prio`, Thor would introduce a variable k_b recording the length of the linked list starting from `buffer`. At the command `buffer = NULL`, we initialize k_b to zero. At the lines `prev->next = x` within `sorted_insert`, the length of that linked list is increased; therefore the abstraction will increment k_b. Similarly, Thor will introduce another variable k_c recording the length of the linked list from `c`. Corresponding to the assignment `c=buffer`, the abstraction will set k_c=k_b, and at the assignment `c=c->next`, the abstraction decrements k_c. Also, when we exit the `while(c!=NULL)` loop, we know that `c==0`, and hence also k_c=0.

Fig. 5.2 shows the control-flow graph (CFG) of the resulting abstraction of `prio`. The CFG contains three nodes corresponding to the three loops in the `prio` function. These nodes are connected by the edges which are annotated with the code occurring between the locations. The transitions between locations come in two forms: assignments v=e`;` and assumption checks `[`e`];`. The assumptions prune executions in which the condition e does not hold.

For brevity, calls to the function `sorted_insert` in Fig. 5.2 have been summarized as the transition $\{\mathtt{k_b}\texttt{++;h++;}\}$ from location 7 to 7, but our technique is designed to work for a fully expanded CFG of the code.

Numerical bounds analysis

Numerical heap abstraction produces program in Fig. 5.2 over the variables $\mathtt{n}$, h, $\mathtt{k}$, $\mathtt{k_b}$, and $\mathtt{k_c}$. The only parameter is the variable $\mathtt{n}$. We consider a template map η^T that assigns to each program location a conjunction of three linear inequalities. For example, for the location ℓ_7 we have

$$\begin{aligned}\eta^T(\ell_7): \quad & \alpha_{\mathtt{n}}\mathtt{n} + \alpha_h h + \alpha_{\mathtt{k}}\mathtt{k} + \alpha_{\mathtt{k_b}}\mathtt{k_b} + \alpha_{\mathtt{k_c}}\mathtt{k_c} \leq \alpha \wedge \\ & \beta_{\mathtt{n}}\mathtt{n} + \beta_h h + \beta_{\mathtt{k}}\mathtt{k} + \beta_{\mathtt{k_b}}\mathtt{k_b} + \beta_{\mathtt{k_c}}\mathtt{k_c} \leq \beta \wedge \\ & \gamma_{\mathtt{n}}\mathtt{n} + \gamma_h h + \gamma_{\mathtt{k}}\mathtt{k} + \gamma_{\mathtt{k_b}}\mathtt{k_b} + \gamma_{\mathtt{k_c}}\mathtt{k_c} \leq \gamma\end{aligned}$$

The bound template at this location is

$$Bnd^T(\ell_7): \quad h \leq \delta_{\mathtt{n}}\mathtt{n} + \delta \ .$$

Next, BoundGen creates a conjunction of constraints Ψ over the template parameters from all program locations. We only present two constraints from Ψ that are created at lines 3 and 5 for the location ℓ_7 and the loop transition at the location ℓ_7 respectively. The first constraint is the implication

$$\forall \mathtt{n}\ \forall h\ \forall \mathtt{k}\ \forall \mathtt{k_b}\ \forall \mathtt{k_c} : \eta^T(\ell_7) \rightarrow Bnd^T(\ell_7) \ .$$

The second constraint involves the transition relation of the loop:

$$\begin{aligned}&\forall \mathtt{n}\ \forall h\ \forall \mathtt{k}\ \forall \mathtt{k_b}\ \forall \mathtt{k_c}\ \forall \mathtt{n}'\ \forall h'\ \forall \mathtt{k}'\ \forall \mathtt{k_b}'\ \forall \mathtt{k_c}' : \\ &\quad (\eta^T(\ell_7) \wedge \mathtt{k} < \mathtt{n} \wedge \mathtt{n}' = \mathtt{n} \wedge h' = h + 1 \wedge \mathtt{k}' = \mathtt{k} + 1 \wedge \mathtt{k_b}' = \mathtt{k_b} + 1 \wedge \mathtt{k_c}' = \mathtt{k_c}) \rightarrow \quad \eta^T(\ell_7)'\end{aligned}$$

We solve Ψ and obtain $\delta_{\mathtt{n}} = 1$ and $\delta = 0$ for the bound template parameters occurring in the location ℓ_7, *i.e.*, we have

$$Bnd^T(\ell_7) = (h \leq \mathtt{n}) \ .$$

The corresponding invariant map assigns $h \leq \mathtt{k_b} \wedge \mathtt{k_b} \leq \mathtt{k} \wedge h \leq \mathtt{n}$ to the location ℓ_7. In our example, the bound occurs in the corresponding inductive invariant; in general, however, this need not be the case.

In the algorithm from Fig. 4.2 we start with a candidate bound $h \leq 0$ at each location. We can then attempt to prove that $h \leq 0$ at every location using a symbolic model checker (this corresponds to lines 5-7 of Fig. 4.2. In this case $\mathtt{h} \leq 0$ is not necessarily true at location 7 in Fig. 5.2, in which case the symbolic model checker will return a witness counterexample path. Imagine that we get the path $\pi = 4 \rightarrow 7$. In this case PathProgram(π) will return a sub-program of Fig. 5.2, as found in Fig. 5.3. We can then find a bound on this sub-program, resulting in $\mathtt{h} \leq \mathtt{n}$. Thus, we refine the candidate whole-program bound to be $\mathtt{h} \leq 0 \vee \mathtt{h} \leq \mathtt{n}$. Repeating the steps from lines 5-7 allows us to prove that $\mathtt{h} \leq 0 \vee \mathtt{h} \leq \mathtt{n}$ is a valid bound for the whole program. After simplification, we return $\mathtt{h} \leq \mathtt{n}$.

Array-based heap management

Numerical boundedness analysis computes a bound on the maximal amount of memory that is dynamically allocated during program computation, and represents this bound as a function of generic parameters. When

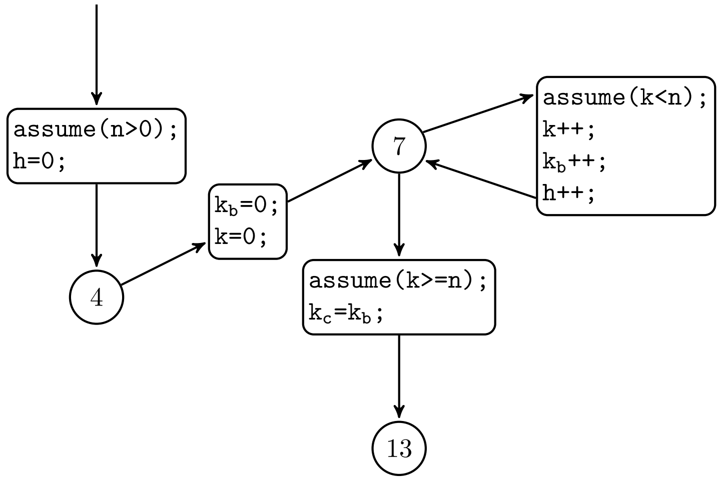


Figure 5.3: Path program for the program from Fig. 5.2 and a path consisting of transitions between the locations (ℓ_{init}, ℓ_4), (ℓ_4, ℓ_7), (ℓ_7, ℓ_7), and (ℓ_7, ℓ_{13}).

synthesizing a hardware implementation, the generic parameters are instantiated. Hence we obtain a concrete bound, say N.

Next, we replace all heap operations in the program $\mathcal{P}$ by operations on a statically allocated array `a` of size N. Each pointer to the heap becomes an array index. Field accesses are converted into arithmetic operations over array indices. For example, the statement `c = c->next;` from the program in Fig. 5.1 becomes `c = a[c+4];`, where the offset 4 is due to the four byte size of an array cell.

We use a list of array indices that is embedded into the array `a` to keep track of free array cells. Each list element is an index of a free cell. We introduce a global variable `m` that stores the head of the list, and hence the cell at index `m` is free. Then, the value of `a[m]` is the next list element, which is the index of the second free cell stored in the list. We obtain the third element by accessing `a[a[m]]` and so on. Initially $\mathtt{m} = 0$ and the array `a` is initialized in the following way:

$$\forall 0 \leq i < N : \mathtt{a}[i] = i + 1 \ .$$

A call to `malloc()` consumes the head of the list. That is, `x = malloc()` is implemented by the sequence of instructions `x = m; m = a[m];`, where the first assignment delivers the free cell and the second assignment ensures that the subsequent call to `malloc` will return the next free cell in the list. We do not need to check whether the free list empty because the boundedness analysis guarantees that it will never happen, i.e., we have $m \leq N$.

Fig. 5.4 illustrates the array-based treatment of `malloc`. We assume that the heap stores data structure `LINK`, whose size is two integers, and that each array cell is of size one integer. The array on the left is free starting at the index 7, as represented by the valuation $\mathtt{m} = 7$, $\mathtt{a}[7] = 9$, etc. After executing `x = malloc(2);`, assigning `x->data = 12;`, the cell at index 7 is no longer free. It stores the data value 12. The next free cell becomes the first one available, i.e., we have $\mathtt{m} = 9$. After identifying the predecessor and successor of `x`, i.e., inserting x into the sorted heap, we obtain the array shown on the right in Fig. 5.4.

A call to `free(x)` pushes `x` onto the free list. That is, this call translates to a pair of statements `a[x] = m; m = x;`. The last freed cell will be the first free cell in the list of free cells, i.e., the subsequent call to `malloc` will return the last freed cell.

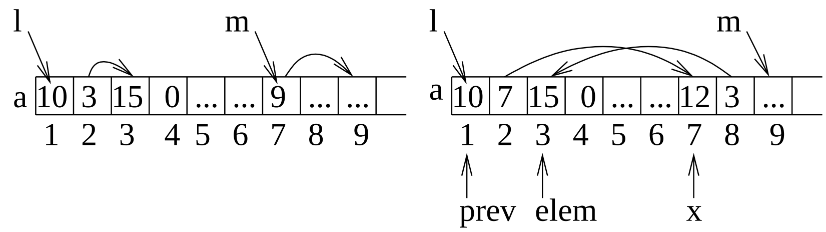


Figure 5.4: Creation of a new `LINK` structure in the array-based heap implementation.

5.3 Experimental results

In this section we discuss the results of our experiments with the proposed synthesis procedure on a number of real-world examples. Before discussing the outputs of our tool, we first describe the problems solved by the C-based software models.

Priority queue – This is our running example from Figure 5.1. The design has one input signal and one output signal. The implementation repeatedly inputs n elements, sorts them, and outputs them in a sorted order. For the sake of experimental evaluation we chose $n = 10$.

Merge sort – This example implements a merger of two sorted sequences. The design has two input signals and one output signal. The implementation repeatedly receives n_1 sorted elements through the first input signal and n_2 sorted elements through the second input signal. Using the merge sort it combines the two sequences into one sorted sequence, which is then output. For the sake of experimental evaluation we chose $n_1 = 10$ and $n_2 = 10$.

Packet sorting – This example implements a simple network element. The design has two input signals and one output signal. The implementation repeatedly inputs packet data through the first input signal and packet identifier through the second input signal. It inserts these packets into a buffer while ignoring duplicate identifiers, until it fills a buffer with n packets. It then sorts the received packets by their identifier and outputs them. For the sake of experimental evaluation we chose $n = 10$.

Binary search tree dictionary – This example implements a data structure for storing a set of elements with a test for membership. The design has two input signals and one output signal. The implementation repeatedly inputs n_1 elements through the first input signal and builds a binary search tree out of them. This is followed by receiving n_2 queries through the second input signal and producing the correct response through the output signal. For the sake of experimental evaluation we chose $n_1 = 10$ and $n_2 = 10$.

Each of these models was succesfully run through the sequence of procedures described in this paper: shape analysis, bounds analysis, and array transformation.

Table 5.1 lists the symbolic bounds for our examples in bytes.[1] These symbolic bounds were then concretized using the aforementioned values and run through our translation tool which inputs a C program and a concrete bound and generates a functionally equivalent VHDL program. Table 5.1 also lists lines of code (LOC) for both the hand-written C models and their automatically generated VHDL counterparts. The running time ranges from minutes to hours depending on the example.

Our VHDL generation step is carefully crafted to work well with FPGA synthesis tools. The generated VHDL files were synthesized using the Altera Quartus II 9.0 tools (build 184 04/29/2009 SP1 SJ Web Edition) targeting Stratix III FPGAs. The results are shown in Table 5.2. The ALUT (Altera's adaptive look-up tables) column gives an indication of the size of the combinational elements in the generated design. The registers column indicates how many flip-flops in the logic fabric were used for registers. The block

[1] The size of data types and structure alignment of a 32-bit architecture (e.g. 4-byte pointers) is assumed.

Program	Bound	C LOC	VHDL LOC	Bound synthesis time
merge	$8 * n_1 + 8 * n_2$	80	1927	600m
prio	$8 * n$	56	1475	4s
packet	$12 * n + 8$	95	2430	6s
bst_dict	$24 * n_1$	142	2703	80s

Table 5.1: Computed bounds and lines of code.

Program	ALUTs	Registers	Block Mem	Blocks	Speed
merge	5,157	4,694	8,192	2	90MHz
prio	5,859	4,598	4,096	1	83MHz
packet	9,413	9,158	8,192	2	76MHz
bst_dict	5,786	5,660	8,192	2	125MHz

Table 5.2: Synthesis and implementation results.

mem column indicates how many memory bits in the generated design were implemented using embedded memory blocks and the following column shows how many independent memories were synthesized. The last column shows the maximum speed. In all cases the tools automatically picked the smallest EP3SL50F484C2 FPGA and package and the timing results are given for this part.

Most of the synthesized circuits occupy only a small portion of the smallest Stratix-III FPGA. The largest design is packet which utilizes 25% of the combinational ALUTs but less than 1% of the available block memory and only 24% of the available logic registers. The smallest design is prio which occupies 15% of the available combinational ALUTs, 12% of the available logic registers and less than 1% of the available block memory. The operating frequency of these circuits is in a range which is typical for FPGA circuits used as co-processing circuits. We have tested several of our examples running on a Cyclone II FPGA on the Altera DE2 board. For example, the priority encoder circuit was synthesized, implemented and run on the Altera Cyclone II EP2C35F672C6 FPGA (supporting 33,216 logic elements) and we have observed the correct behavior on actual hardware using the SignalTap logic analyzer. Our conclusion from these preliminary results is that we have identified a viable approach for translating heap-based C programs into VHDL designs which have acceptable area utilization and performance.

Our bounds computation algorithm was able to compute useful bounds. However, at the moment we do not have enough experimental data to provide an thorough estimate for the quality of bounds computation.

Examples of failure Our approach for symbolic bounds synthesis can fail in various ways. For example, the input program might operate over DAGs (*e.g.* BDDs) or hash tables; in which case, we would currently fail to produce an arithmetic abstraction. Note that—even in the case of programs with simple linked data structures—improving the scalability and accuracy of shape analysis is an area of active research. When we successfully generate arithmetic abstractions, our constraint-based synthesis algorithm can also fail. The abstraction may be too coarse, or the problem may be too complex (*e.g.* highly non-linear).

Part II

Constraint solving algorithms

Chapter 6

Introduction to interpolation

For many path based constraint generation and solving methods of verification [66], the significant cost of verification goes into solving interpolation constraints obtained from symbolic execution of program paths. An efficient interpolation procedure can significantly reduce the verification time. In this chapter, we present interpolation and an interpolation procedure for the theory of linear arithmetic and uninterpreted functions.

6.1 Interpolation

We are going to use the following result in the first order logic.

Theorem 4 (Craig interpolation theorem [20]). *Let A and B be first order logical formulas such that $A \to B$ is valid. Then, there exist a first order logical formula I containing only predicate symbols, function symbols and constants occurring in both A and B such that $A \to I$ and $I \to B$ are valid.*

The above theorem is not in the form in which the above theorem is used in program verification. The following equivalent theorem captures the need of over-approximating reachable program states at an intermediate point of an infeasible program path.

Theorem 5. *Let A and B be first order logical formulas such that $A \wedge B \to \mathit{false}$ is valid. Then, there exist a first order logical formula I containing only predicate symbols, function symbols and constants occurring in both A and B such that $A \to I$ and $I \wedge B \to \mathit{false}$ are valid. I is called an* interpolant *of A and B.*

When verifying programs, we are interested in computing interpolants in a given first order theory and for a given class of formulas. Let $\mathcal{T}$ be a theory with signature Σ, and $\Sigma_{\mathcal{T}} \subseteq \Sigma$ be the set of symbols that are interpreted in $\mathcal{T}$. All the other symbols in $\Sigma \setminus \Sigma_{\mathcal{T}}$ are considered to be uninterpreted. A class of Σ-formulas is a subset of all Σ-formulas. For a Σ-formula A, $\models_{\mathcal{T}} A$ denotes that A is true in all models of $\mathcal{T}$. For a term t, let $\mathit{Smb}(t)$ be the set of uninterpreted symbols and free variables appearing in t. Smb is canonically extended to formulas and sets of formulas.

Definition 1 (Theory specific interpolant [85]). *Let $\mathcal{C}$ and $\mathcal{C}^I$ be classes of Σ-formulas. Let formulas A and B in $\mathcal{C}$ such that $\models_{\mathcal{T}} A \wedge B \to \mathit{false}$. We say that formula I is* theory specific interpolant *of A and B if (1) $\models_{\mathcal{T}} A \to I$, (2) $\models_{\mathcal{T}} I \wedge B \to \mathit{false}$, (3) $I \in \mathcal{C}^I$, and (4) $\mathit{Smb}(I) \subseteq \mathit{Smb}(A) \cap \mathit{Smb}(B)$.*

The requirement of $I \in \mathcal{C}^I$ implies that a theory specific interpolant may not exist.

We will present an algorithm for computing theory specific interpolants for the combined theory of linear arithmetic and uninterpreted functions. The algorithm depends on a notion of partial interpolants, which is more general concept than theory specific interpolants.

Definition 2 (F-partial interpolant [85]). *Let $\mathcal{C}$ and $\mathcal{C}^I$ be classes of Σ-formulas. Let F be a Σ-formula. Let A and B be formula in $\mathcal{C}$ such that $\models_{\mathcal{T}} A \wedge B \to F$. We say that formula I is F-*partial interpolant *of A and B if (1) $\models_{\mathcal{T}} A \to I$, (2) $\models_{\mathcal{T}} I \wedge B \to F$, (3) $I \in \mathcal{C}^I$, and (4) $I \subseteq (\mathit{Smb}(A) \cap \mathit{Smb}(B)) \cup \mathit{Smb}(F)$.*

If $\models_{\mathcal{T}} F \to \mathit{false}$ and $\mathit{Smb}(F) = \emptyset$ then F-partial interpolant is a theory specific interpolant.

6.2 Proof rules and proof trees for $\mathcal{T}_{\mathbf{LI+UIF}}$

Our interpolation algorithm relies on unsatisfiability proofs [65]. We use a standard set of proof rules for the combination of linear arithmetic and uninterpreted functions. The implementation of the corresponding proof search procedure is irrelevant for the interpolation algorithm, yet we assume that this procedure is complete and use an existing tool for this task, e.g. [11, 21, 52]. In Chapter 7, we will present an implementation of a proof search procedure based on simplex.

For a conjunctive constraint C, Figure 6.1 presents the proof rules. The rule PHyp states that atoms appearing in C are provable from C. The rule PComb infers that a set of inequalities implies a positively weighted sum thereof. The congruence rule PCong represents a form of the functionality axiom that states that equal inputs to a function lead to equal results. We are only interested in one inequality part of this axiom. The side condition of PCong is taken from the interpolating proof rules of [65], and simplifies the proof tree annotation in a way similar to [65].

Proof tree

A proof tree is produced by applying the proof rules and inferring atomic formulas. We assume that there exists a mechanism that uniquely identifies the nodes of the proof tree, even in the presence of nodes that are labeled by equal atoms, for example by numbering them. For clarity of exposition, we omit any details of such mechanism and assume that the node label carries all necessary information.

Formally, a label l is an application of a proof rule defined as

$$\begin{array}{lll} \text{labels} & \ni l & ::= \text{PH{\scriptsize YP}} \mid \text{PC{\scriptsize OMB}}(c, \dots, c) \mid \text{PC{\scriptsize ONG}} \\ \text{labeled edges} & \ni e & ::= (A, l, (A, \dots, A)). \end{array}$$

Recall A is an atom and c is a rational number in $\mathcal{T}_{\text{LI+UIF}}$ (section 2.2).

A proof tree P is a finite subset of labeled edges. For each $(t \leq 0, l, (t_1 \leq 0, \dots, t_n \leq 0)) \in P$, $t \leq 0$ is called a parent node, l is label of the parent node, and $t_1 \leq 0, \dots, t_n \leq 0$ are called child nodes. A proof tree P is *inferred* from a conjunctive constraints C if

- $\forall (t \leq 0,\ \text{PH{\scriptsize YP}},\ ()\ ) \in P\ :\ \ t \leq 0 \in C$,
- $\forall (t \leq 0, \text{PC{\scriptsize OMB}}(\lambda_1, \dots, \lambda_n),\ (t_1 \leq 0, \dots, t_n \leq 0)) \in P :$
  $t = \lambda_1 t_1 + \dots + \lambda_n t_n \wedge \lambda_1, \dots, \lambda_n > 0 \wedge \forall i \in 1..n : (t_i \leq 0, \_, \_) \in P$, and
- $\forall (f(t_1, \dots, t_n) - f(s_1, \dots, s_n) \leq 0, \text{PC{\scriptsize ONG}}, (t_1 - s_1 \leq 0, s_1 - t_1 \leq 0, \dots, t_n - s_n \leq 0, s_n - t_n \leq 0)) \in P\ :$
  $(\forall i \in 1..n : (t_i - s_i \leq 0, \_, \_) \in P \wedge (s_i - t_i \leq 0, \_, \_) \in P)$.

For a proof tree P, a conjunctive constraint C, and an atom $t \leq 0$, P *proves* $\models C \rightarrow t \leq 0$ if P is inferred from C, and contains $(t \leq 0, \_, \_)$. If a proof tree P proves $\models C \rightarrow 1 \leq 0$ then C is unsatisfiable.

$$\text{PH{\scriptsize YP}}\ \frac{}{t \leq 0}\ t \leq 0 \in \mathit{atoms}(C) \qquad\qquad \text{PC{\scriptsize OMB}}\ \frac{t_1 \leq 0 \quad \dots \quad t_n \leq 0}{\lambda_1 t_1 + \dots + \lambda_n t_n \leq 0}\ \lambda_1, \dots, \lambda_n > 0$$

$$\text{PC{\scriptsize ONG}}\ \frac{\begin{array}{cc} t_1 - s_1 \leq 0 & s_1 - t_1 \leq 0 \\ \vdots & \vdots \\ t_n - s_n \leq 0 & s_n - t_n \leq 0 \end{array}}{f(t_1, \dots, t_n) - f(s_1, \dots, s_n) \leq 0}\ f(t_1, \dots, t_n), f(s_1, \dots, s_n) \in \mathit{subterms}(C)$$

Figure 6.1: Standard [65], complete proof rules PHyp, PComb, and PCong for the combination of linear rational/real arithmetic and uninterpreted functions. C is a conjunction of atoms.

$$\textsc{PComb}\ \dfrac{\textsc{PCong}\ \dfrac{\textsc{PHyp}\ \dfrac{}{y - x \leq 0} \qquad \textsc{PHyp}\ \dfrac{}{x - y \leq 0}}{f(y) - f(x) \leq 0} \qquad \textsc{PHyp}\ \dfrac{}{f(x) - f(y) + 1 \leq 0}}{1 \leq 0}$$

Figure 6.2: An example of a proof tree.

Example 2. Consider the following conjunctive constraint.

$$y - x \leq 0 \wedge x - y \leq 0 \wedge f(x) - f(y) + 1 \leq 0$$

Figure 6.2 presents a proof tree inferred from the above conjunctive constraint. This proof tree proves that the above conjunction is unsatisfiable.

6.3 Algorithm for interpolation in $\mathcal{T}_{\mathbf{LI+UIF}}$

Our algorithm computes a theory specific interpolant of conjunctive constraints A and B by annotating a proof tree that proves $\models A \wedge B \rightarrow 1 \leq 0$. The algorithm annotates each node $t \leq 0$ with a $t \leq 0$-partial interpolant. Each partial interpolant is of the following form, called *solution constraints*.

$$\text{solution constraints} \ni S ::= t \leq 0 \mid C \wedge (C \rightarrow S)$$

Recall C is a conjunctive constraint in $\mathcal{T}_{\mathrm{LI+UIF}}$ (section 2.2). To simplify the presentation, we write a solution constraint

$$C_1 \wedge (D_1 \rightarrow \ldots C_r \wedge (D_r \rightarrow p \leq 0) \ldots)$$

as a pair consisting of a corresponding sequence and a term $\langle((C_1, D_1), \ldots, (C_r, D_r)), p\rangle$. A solution constraint $p \leq 0$, i.e. when $r = 0$, is represented by $\langle(), p\rangle$.

Given the proof tree that proves $\models A \wedge B \rightarrow 1 \leq 0$, we annotate its nodes with partial interpolants using the rules shown in Figure 6.3. For each node $t \leq 0$, the annotation is $t \leq 0$-partial interpolant in the form of solution constraints and is enclosed by a pair of square brackets. These rules apply different annotations for different cases of antecedents of each rule from Figure 6.1. So rules PHyp and PCong lead to multiple annotation rules. AHyp-A annotates a leaf node $t \leq 0$ if $t \leq 0 \in A$. AHyp-B annotates a leaf node $t \leq 0$ if $t \leq 0 \in B$. The rule AComb annotates a parent node when provided with the annotations of its children in case when the parent was obtained by a positively weighted sum. The congruence rule PCong leads to four annotation rules ACong-BB, ACong-BB, ACong-BB, and ACong-BB. These rules annotate parent nodes obtained by applications of the congruence rule depending on where antecedents of the congruence come from.

The annotation of the node $1 \leq 0$ is a theory specific interpolant of A and B.

6.4 Correctness

We need to prove that annotations of proof rules are partial interpolants. We will demonstrate that our annotation algorithm maintains the following invariant at each node of proof tree.

Definition 3 ($t \leq 0$-annotation invariant)**.** *Let $t \leq 0$ be an atom and let A and B be conjunctive constraints such that $\models A \wedge B \rightarrow t \leq 0$. A solution constraint $\langle((C_1, D_1), \ldots, (C_r, D_r)), p\rangle$ satisfies $t \leq 0$-annotation invariant if*

(1) for each $k \in 1..r$, $\models A \wedge \bigwedge_{i=1}^{k-1} D_i \rightarrow C_k$,

(2) for each $k \in 1..r$, $\models B \wedge \bigwedge_{i=1}^{k} C_i \rightarrow D_k$,

$$\textsc{AHyp-A}\ \frac{}{t \leq 0\ [\langle (), t \rangle]}\ t \leq 0 \in A$$

$$\textsc{AHyp-B}\ \frac{}{t \leq 0\ [\langle (), 0 \rangle]}\ t \leq 0 \in B$$

$$\textsc{AComb}\ \frac{t_1 \leq 0\ [\langle L_1, p_1 \rangle] \quad \dots \quad t_n \leq 0\ [\langle L_n, p_n \rangle]}{\lambda_1 t_1 + \dots + \lambda_n t_n \leq 0\ [\langle L_1 \bullet \dots \bullet L_n, \lambda_1 p_1 + \dots + \lambda_n p_n \rangle]}\ 0 < \lambda_1, \dots, \lambda_n$$

$$\textsc{ACong-BB}\ \frac{\begin{array}{cc} t_1 - s_1 \leq 0\ [\langle L_1, p_1 \rangle] & s_1 - t_1 \leq 0\ [\langle L'_1, p'_1 \rangle] \\ \vdots & \vdots \\ t_n - s_n \leq 0\ [\langle L_n, p_n \rangle] & s_n - t_n \leq 0\ [\langle L'_n, p'_n \rangle] \end{array}}{\begin{array}{l} f(t_1, \dots, t_n) - f(s_1, \dots, s_n) \leq 0 \\ [\langle\ L_1 \bullet \dots \bullet L_n \bullet L'_1 \bullet \dots \bullet L'_n \bullet \bigwedge_{i=0}^{n}(p_i \leq 0 \wedge p'_i \leq 0), \mathit{true}),\ 0\ \rangle] \end{array}}\ \begin{array}{l} \mathit{Smb}(f(t_1, \dots, t_n)) \subseteq \mathit{Smb}(B) \\ \mathit{Smb}(f(s_1, \dots, s_n)) \subseteq \mathit{Smb}(B) \end{array}$$

$$\textsc{ACong-AB}\ \frac{\begin{array}{cc} t_1 - s_1 \leq 0\ [\langle L_1, p_1 \rangle] & s_1 - t_1 \leq 0\ [\langle L'_1, p'_1 \rangle] \\ \vdots & \vdots \\ t_n - s_n \leq 0\ [\langle L_n, p_n \rangle] & s_n - t_n \leq 0\ [\langle L'_n, p'_n \rangle] \end{array}}{\begin{array}{l} f(t_1, \dots, t_n) - f(s_1, \dots, s_n) \leq 0 \\ [\langle L_1 \bullet \dots \bullet L_n \bullet L'_1 \bullet \dots \bullet L'_n \bullet \\ \quad (\bigwedge_{i=0}^{n} p_i + p'_i \leq 0, \bigwedge_{i=0}^{n} -p_i - p'_i \leq 0), \\ \quad f(s_1 + p_1, \dots, s_n + p_n) - f(s_1, \dots, s_n)\ \rangle] \end{array}}\ \begin{array}{l} \mathit{Smb}(f(t_1, \dots, t_n)) \subseteq \mathit{Smb}(B) \\ \mathit{Smb}(f(s_1, \dots, s_n)) \nsubseteq \mathit{Smb}(B) \end{array}$$

$$\textsc{ACong-BA}\ \frac{\begin{array}{cc} t_1 - s_1 \leq 0\ [\langle L_1, p_1 \rangle] & s_1 - t_1 \leq 0\ [\langle L'_1, p'_1 \rangle] \\ \vdots & \vdots \\ t_n - s_n \leq 0\ [\langle L_n, p_n \rangle] & s_n - t_n \leq 0\ [\langle L'_n, p'_n \rangle] \end{array}}{\begin{array}{l} f(t_1, \dots, t_n) - f(s_1, \dots, s_n) \leq 0 \\ [\langle\ L_1 \bullet \dots \bullet L_n \bullet L'_1 \bullet \dots \bullet L'_n \bullet \\ (\bigwedge_{i=0}^{n} p_i + p'_i \leq 0, \bigwedge_{i=0}^{n} -p_i - p'_i \leq 0), \\ f(t_1, \dots, t_n) - f(t_1 + p'_1, \dots, t_n + p'_n)\ \rangle] \end{array}}\ \begin{array}{l} \mathit{Smb}(f(t_1, \dots, t_n)) \nsubseteq \mathit{Smb}(B) \\ \mathit{Smb}(f(s_1, \dots, s_n)) \subseteq \mathit{Smb}(B) \end{array}$$

$$\textsc{ACong-AA}\ \frac{\begin{array}{cc} t_1 - s_1 \leq 0\ [\langle L_1, p_1 \rangle] & s_1 - t_1 \leq 0\ [\langle L'_1, p'_1 \rangle] \\ \vdots & \vdots \\ t_n - s_n \leq 0\ [\langle L_n, p_n \rangle] & s_n - t_n \leq 0\ [\langle L'_n, p'_n \rangle] \end{array}}{\begin{array}{l} f(t_1, \dots, t_n) - f(s_1, \dots, s_n) \leq 0 \\ [\langle\ L_1 \bullet \dots \bullet L_n \bullet L'_1 \bullet \dots \bullet L'_n \bullet \\ (\mathit{true}, \bigwedge_{i=0}^{n}(t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p'_i \leq 0)), \\ f(t_1, \dots, t_n) - f(s_1, \dots, s_n)\ \rangle] \end{array}}\ \begin{array}{l} \mathit{Smb}(f(t_1, \dots, t_n)) \nsubseteq \mathit{Smb}(B) \\ \mathit{Smb}(f(s_1, \dots, s_n)) \nsubseteq \mathit{Smb}(B) \end{array}$$

Figure 6.3: Annotation rules for computing an interpolant of A and B [65].

(3) $\models A \wedge \bigwedge_{i=1}^{r} D_i \rightarrow p \leq 0$,

(4) $\models B \wedge \bigwedge_{i=1}^{r} C_i \rightarrow t - p \leq 0$,

(5) $Smb(\{C_1, \ldots, C_r, D_1, \ldots, D_r, p \leq 0\}) \subseteq Smb(A)$, *and*

(6) $Smb(\{C_1, \ldots, C_r, D_1, \ldots, D_r, t - p \leq 0\}) \subseteq Smb(B)$.

The following theorems entail the correctness of the interpolation procedure, i.e., the annotation of the node $1 \leq 0$ is a theory specific interpolant of conjunctive constraints A and B.

Theorem 6. *If a solution constraint* $I = \langle((C_1, D_1), \ldots, (C_r, D_r)), p\rangle$ *satisfies* $t \leq 0$*-annotation invariant then* I *is a* $t \leq 0$*-partial interpolant.*

Theorem 7. *Annotation rules in Figure 6.3 compute annotations that satisfy Definition 3.*

The algorithm for computing interpolants is a special case of a algorithm for solving Horn clauses. So we defer proofs of the above theorems until Chapter 8.

Chapter 7

Proof producing CLP(LI+UIF)

CLP(Q) [52] is a useful building block for verification tools. However, CLP(Q) does not currently produce proofs, which are needed to compute interpolants, and does not deal with the theory of uninterpreted functions, which is useful for modelling complex operations when verifying programs. In this chapter, we present a tool CLP(LI+UIF) that checks unsatisfiability of conjunctive constraints in the theory of linear arithmetic and uninterpreted functions and also produces a proof tree when unsatisfiability is detected.

The existing simplex based proof producing algorithms [14, 23] use a version of simplex that does not apply constant propagation. These algorithms construct proofs by relying on an instrumentation of the input constraints. This instrumentation leads to the creation of many additional variables. In this chapter, we present an alternative proof producing simplex based algorithm that relies on an instrumentation of an incremental, constant propagating simplex. Our instrumentation does not require incremental simplex to introduce additional variables for proof construction and does not prohibit constant propagation.

In following sections, first we will present the algorithm used in CLP(Q) and its extension for supporting uninterpreted functions. Second, we will discuss the incompatibility of existing algorithms for proof tree generation with CLP(Q). Third, we will present our instrumentation of the algorithm in CLP(Q).

7.1 CLP(Q)

CLP(Q) [52] is a linear programming tool. Since, We are only interested in the unsatisfiability of conjunctive constraints, we will only consider *phase 1* of simplex. In CLP(Q), this phase is implemented as a version of incremental simplex.

The incremental simplex takes as input a sequence of linear atoms. At any instant, the input so far is stored in a so called *solved form* that represents the input in a normal form. Given the next input from the input sequence and the current solved form, the incremental simplex computes the next solved form. A solved form of a conjunctive constraint exists if and only if the conjunctive constraint is satisfiable. Therefore, failure to compute a solved form indicates that the input considered so far is unsatisfiable. In practice, the incremental simplex is more efficient than a non-incremental one for satisfiablity checking [53].

The algorithm of the CLP(Q) solver is described in [51] and is an optimized version of algorithm presented in [70]. We will now reformulate this algorithm in the notation that is convenient to us. The CLP(Q) solver has the following important optimizations that affect proof tree extraction.

- The CLP(Q) solver avoids introduction of as many slack variables as possible.

- If the input implies that a variable is equal to a constant then the CLP(Q) solver replaces this variable with the constant.

Solved form

In general, a solved form is a variant of the standard simplex tableau [77]. There are various kinds of solved forms with different properties regarding data structure representation, ability to detect equalities and disequalities, and efficiency of transforming a conjunctive constraints into the solved form [53]. The CLP(Q) uses a solved form in which equality detection is most efficient and therefore the treatment of disequalities is trivial, but the cost of computing the solved form is high. Equality detection is also very significant for us since congruence checker for uninterpreted functions depends on equality detection.

Formally, the solved form in CLP(Q) is a tuple $(X, \mathit{Basis}, \mathit{Def}, \mathit{Low}, \mathit{Up}, \mathit{Active}, \mathit{Val})$ where

- $X = \{x_1, \ldots, x_n\}$ is a finite ordered set of rational variables,
- $\mathit{Basis} \subseteq X$ is called a *basis*,
- $\mathit{Def} : X \to$ linear terms, Def assigns definitions to variables and we require that for each $k \in 1..n$,

$$\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j,$$

  where $c \in \mathbb{Q}$, $J \subseteq 1..n$ and $\forall j \in J.\ c_j \in \mathbb{Q} \setminus \{0\}$,
- $\mathit{Low} : X \to \mathbb{Q} \cup \{-\infty\}$, Low defines lower bounds on variables,
- $\mathit{Up} : X \to \mathbb{Q} \cup \{+\infty\}$, Up defines upper bounds on variables,
- $\mathit{Active} : X \to \{\mathtt{none}, \mathtt{lower}, \mathtt{upper}\}$,
- $\mathit{Val} : X \to \mathbb{Q}$,
- and the conditions listed below are satisfied.

Let $k \in 1..n$. x_k is *undefined* if $\mathit{Def}(x_k) = x_k$ and is *defined* otherwise. x_k is *unbounded* if $\mathit{Low}(x_k) = -\infty$ and $\mathit{Up}(x_k) = +\infty$, otherwise x_k is *bounded.* x_k is *active* if $\mathit{Active}(x_k) \neq \mathtt{none}$, and is *inactive* otherwise. A solved form must satisfy the following conditions:

(1) $\mathit{Def}(x_k)$ only contains undefined variables.

(2) If $x_k \in \mathit{Basis}$ then x_k is defined, bounded, inactive, and all variables appearing in $\mathit{Def}(x_k)$ are active.

(3) If $x_k \notin \mathit{Basis}$ and x_k is defined then x_k is unbounded and inactive.

(4) If x_k is active then x_k is bounded, undefined, and there is $x_b \in \mathit{Basis}$ such that x_k occurs in $\mathit{Def}(x_b)$.

(5) $\mathit{Low}(x_k) < \mathit{Up}(x_k)$.

(6) If $\mathit{Active}(x_k) = \mathtt{lower}$ then $\mathit{Low}(x_k) \neq -\infty$, and if $\mathit{Active}(x_k) = \mathtt{upper}$ then $\mathit{Up}(x_k) \neq +\infty$.

(7) If x_k is undefined then

$$\mathit{Val}(x_k) = \begin{cases} \mathit{Low}(x_k) & \text{if } \mathit{Active}(x_k) = \mathtt{lower}, \\ \mathit{Up}(x_k) & \text{if } \mathit{Active}(x_k) = \mathtt{upper}, \\ 0 & \text{if } \mathit{Active}(x_k) = \mathtt{none}. \end{cases}$$

(8) If x_k is defined and $\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j$ then $\mathit{Val}(x_k) = c + \sum_{j \in J} c_j \mathit{Val}(x_j)$.

(9) If $x_k \in \mathit{Basis}$ then $\mathit{Low}(x_k) \leq \mathit{Val}(x_k) \leq \mathit{Up}(x_k)$.

Kind	Condition	Basis	Defined	Active	Bounded	Remark
1	$x_k \in \mathit{Basis}$	✓	✓*	×*	✓*	If $\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j$ then $\forall j \in J.\ x_j$ is undefined and active*
2	$x_k \notin \mathit{Basis}$ x_k is defined	×	✓	×*	×*	If $\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j$ then $\forall j \in J.\ x_j$ is undefined*
3	x_k is active	×*	×*	✓	✓*	$\exists x_b \in \mathit{Basis}.\ x_k \in \mathit{Smb}(\mathit{Def}(x_b))$*
4	x_k is undefined x_k is inactive	×*	×	×		

Figure 7.1: Solved form implicitly induces above four kinds of variables. * denotes the property is a restriction imposed by a condition of solved form.

Conditions (1)–(4) impose a syntactic restriction, while (5)–(9) require arithmetic evaluation of solved form. *Def* represents a set of linear equations and condition (1) states that these equations are in a triangular form, which is usually obtained by Gaussian elimination. Conditions (2)–(4) induce four kinds of variables in the solved form that are presented in Figure 7.1. Note that variables of the fourth kind vacuously satisfy (2)–(4), since they violate the respective if-conditions.

A solved form is equivalent to the conjunctive constraint.

$$\bigwedge_{k=1}^{n} (\ x_k = \mathit{Def}(x_k) \wedge \mathit{Low}(x_k) \leq x_k \leq \mathit{Up}(x_k)\ ) \tag{7.1}$$

The conditions (1)–(9) imply that conjunctive constraints in Equation (7.1) are satisfiable. For a given solved form, we can construct a satisfying assignment Val' in following way. We choose assignments for variables in order of first kind, third kind, fourth kind, and second kind.

- For each x_k variable of first or third kind, let $\mathit{Val}'(x_k) = \mathit{Val}(x_k)$.
- Let x_k be a variable of fourth kind. x_k can only appear in the definition of variables of the second kind. The second kind variables are unbounded, therefore, we can choose any value for $\mathit{Val}'(x_k)$ that is between $\mathit{Low}(x_k)$ and $\mathit{Up}(x_k)$.
- Let x_k be a variable of the second kind such that $\mathit{Def}(x_k) = c + \sum_{j \in J} c_j x_j$. We have assigned Val' map for all the undefined variables therefore we can evaluate $\mathit{Def}(x_k)$ under assignments of Val'. Let $\mathit{Val}'(x_k) = c + \sum_{j \in J} c_j\, \mathit{Val}'(x_j)$.

Due to conditions (5)–(9), Val' is a satisfying assignment.

A satisfiable conjunctive constraint can always be transformed into an equisatisfiable solved form. The resulting solved form may contain more variables than the original constraint due to the introduction of slack variables in the process of transformation. The solved form may not be unique.

Example 3 (Solved form)**.** The constraints shown in Figure 7.2(a) are satisfiable. In figure 7.2(b), we show a solved form for the constraints. Variables x_1, x_2, x_3, and x_4 appear in the original constraints. Variables u, v, and w are slack variable that are introduced during the transformation to the solved form. x_4 and x_2 are variables of the first kind. x_3 is variable of the second kind. x_1, u, w, and v are variables of the third kind. There is no fourth kind of variable in this solved form therefore *Val* is satisfying assignment to the original constants.

CLP(Q) algorithm

Figures 7.3, 7.4, and 7.5 present incremental simplex in CLP(Q) [51]. This algorithm takes linear atoms as input sequence. Given an input and the current solved form, CLP(Q) computes the next solved form. If CLP(Q) fails to compute the next solved form then it throws an exception "`Unsatisfiable`".

If the input is an equation then AddEquality is called. If the input is an inequality then AddInequality is called. We refer to these two procedures as *entry* procedures.

$$
\begin{aligned}
x_1 - x_2 + 2x_3 - 2 &\leq 0\\
x_2 - x_1 - x_3 + 3 &\leq 0\\
x_4 + x_3 - 2 &\leq 0\\
2 - x_4 &\leq 0\\
x_1 - 10 &\leq 0\\
2 - x_2 &\leq 0
\end{aligned}
$$

(a)

	Variable	*Def*	*Low*	*Up*	*Active*	*Val*	
basis	x_4	$3 + u + v - w$	2	$+\infty$	`none`	3	defined
basis	x_2	$-4 + x_1 - u - 2v$	2	$+\infty$	`none`	6	defined
	x_3	$-1 - u - v$	$-\infty$	$+\infty$	`none`	-1	defined
	x_1	x_1	$-\infty$	10	`upper`	10	active
slack variables	u	u	0	$+\infty$	`lower`	0	active
slack variables	v	v	0	$+\infty$	`lower`	0	active
slack variables	w	w	0	$+\infty$	`lower`	0	active

(b)

Figure 7.2: (a) An example input to the CLP(Q) solver. (b) Solved form computed by CLP(Q) for the input. x_4 and x_2 are variables of first kind. x_3 is variable of second kind. x_1, u, w, and v are variables of third kind.

Global data structures The global maps X, *Def*, *Low*, *Up*, *Active*, and *Val* are components of the solved form. They are initialized to be empty. So initially the solved form is empty. Note that when we pick a fresh variable x_k at line 11 of AddInequality that means x_k is not referred by any of the input constraints, and x_k is not in current X. During the run of CLP(Q), some variables are detected to be equal to a constant. Such equalities are stored in *queue* and these equalities are added to the solved form at the end of execution of the entry procedures (In AddInequality, lines 13–15 and in AddInequality lines 21–23).

Now we will describe procedures of the algorithm.

Procedures Deref and Initialize Both the entry procedures call Deref to de-reference the term of the input atom. Deref replaces each variable appearing in the input term with its definition in the solved form. If a variable in not yet part of the solved form then the procedure Initialize is called to add the variable in the solved form as a non-basis, undefined, inactive, and unbounded variable. Deref eliminates all defined variables from the input term and returns a term over undefined variables.

Procedure Substitute This procedure takes an undefined variable x_m and a term over other undefined variables as input. Substitute replaces each occurrence of x_m in the definitions by the given input term. These replacements may leads to violation of condition (8). So Substitute also updates *Val* such that condition (8) holds at the end of this procedure. As a result, Substitute turns x_m into a defined variable.

Procedure Pivot The inputs of this procedure are a basis variable x_b, an activation direction *act*, and an undefined and active variable x_i that appears in the definition of x_b. Pivot removes x_b from the basis and adds x_i to the basis using Substitute at lines 1–3. x_b is now an undefined variable that appears in the definition of the basis variable x_i, so x_b has to be made active. Pivot activates x_b in the direction *act* by calling procedure Activate at line 5. Since x_i is added to the basis, x_i is made inactive at line 6.

Procedures Activate and AddBasis Activate activates an inactive variable. It also has to update *Val* to satisfy condition (7) and (8). The inputs of AddBasis are a defined variable x_m and an activation direction *act*. This procedure add x_m to basis and makes it inactive. Each variable x_j appearing in the definition of x_m is activated at lines 4–13. If either of the bounds of x_j does not exist then the other bound is activated at lines 6–9. Otherwise, if c_j is positive then x_j is activated in the direction *act* and if c_j is negative then x_j is activated in the direction opposite to *act* at lines 10–13. $\oplus$ denotes the logical xor operator.

Procedure AddEquality This procedure takes a linear equality $t = 0$ as input. At line 1, t is de-referenced using the solved form. If the solved form implies $t = 0$ then the condition at line 2 is true and procedure continues at line 13. If the condition at line 4 is true then the conjunction of the solved form and

global variables

$\left\{\begin{array}{ll} X = \emptyset & : \text{set of variables} \\ \mathit{Def} = \emptyset & : X \to \text{linear terms} \\ \mathit{Low} = \emptyset & : X \to \mathbb{Q} \cup \{-\infty\} \\ \mathit{Active} = \emptyset & : X \to \{\texttt{none}, \texttt{lower}, \texttt{upper}\} \end{array}\right.$ $\left.\begin{array}{ll} \mathit{Basis} = \emptyset & : \text{set of variables} \\ & \\ \mathit{Up} = \emptyset & : X \to \mathbb{Q} \cup \{+\infty\} \\ \mathit{Val} = \emptyset & : X \to \mathbb{Q} \end{array}\right\}$: solved form

$\mathit{queue} = \emptyset$: set of linear atoms

procedure AddEquality
input
  $t = 0$: linear constraint
begin
  $c + \sum_{j \in J} c_j x_j := \text{Deref}(t)$
  if $c = 0 \wedge J = \emptyset$ **then**
    skip
  elsif $c \neq 0 \wedge J = \emptyset$ **then**
    throw “`Unsatisfiable`”
  elsif $\exists i \in J.\ \mathit{Low}(x_i) = -\infty \wedge \mathit{Up}(x_i) = +\infty$ **then**
    $\text{Substitute}(x_i, -\frac{1}{c_i}(c + \sum_{j \in J \setminus \{i\}} c_j x_j))$
  else
    pick $i \in J$
    $\text{Substitute}(x_i, -\frac{1}{c_i}(c + \sum_{j \in J \setminus \{i\}} c_j x_j))$
    $\text{AddBasis}(x_i, \texttt{lower})$
    $\text{RepairBasis}()$
  if $s = 0 \in \mathit{queue}$ **then**
    $\mathit{queue} := \mathit{queue} \setminus \{s = 0\}$
    $\text{AddEquality}(s = 0)$
end

procedure Initialize
input
  x_i : uninitialized variable
begin
  $X := X \cup \{x_i\}$
  $(\mathit{Def}(x_i), \mathit{Active}(x_i), \mathit{Val}(x_i)) := (x_i, \texttt{none}, 0)$
  $(\mathit{Low}(x_i), \mathit{Up}(x_i)) := (-\infty, +\infty)$
end

procedure AddInequality
input
  $t \leq 0$: linear constraint
begin
  $c + \sum_{j \in J} c_j x_j := \text{Deref}(t)$
  if $c \leq 0 \wedge J = \emptyset$ **then**
    skip
  elsif $c > 0 \wedge J = \emptyset$ **then**
    throw “`Unsatisfiable`”
  elsif $t = a + a_i x_i$ **then**
    $\text{UpdateBound}(a + a_i x_i \leq 0)$
  elsif $J = \{j\}$ **then**
    $\text{UpdateBound}(c + c_j x_j \leq 0)$
  else
    pick fresh x_k (∗ slack variable ∗)
    $\text{Initialize}(x_k)$
    $\mathit{Low}(x_k) := 0$
    if $\exists i \in J. \mathit{Low}(x_i) = -\infty \wedge \mathit{Up}(x_i) = +\infty$ **then**
      $\text{Substitute}(x_i, -\frac{1}{c_i}(c + \sum_{j \in J \setminus \{i\}} c_j x_j + x_k))$
    else
      $\mathit{Def}(x_k) := -(c + \sum_{j \in J} c_j x_j)$
      $\mathit{Val}(x_k) := -(c + \sum_{j \in J} c_j \mathit{Val}(x_j))$
      $\text{AddBasis}(x_k, \texttt{lower})$
      $\text{RepairVar}(x_k)$
  if $s = 0 \in \mathit{queue}$ **then**
    $\mathit{queue} := \mathit{queue} \setminus \{s = 0\}$
    $\text{AddEquality}(s = 0)$
end

procedure Deref
input
  $c + \sum_{j \in J} c_j x_j$: linear term
begin
  $t := c$
  for each $j \in J$ **do**
    if $\mathit{Def}(x_j) = \perp$ **then**
      $\text{Initialize}(x_j)$
    $t := t + c_j \mathit{Def}(x_j)$
  return t
end

procedure Substitute
input
  x_m : undefined variable
  $c + \sum_{j \in J} c_j x_j$: linear term
begin
  $d := c + \sum_{j \in J} c_j \mathit{Val}(x_j) - \mathit{Val}(x_m)$
  for each $x_k \in X$:
  $a + \sum_{i \in I} a_i x_i = \mathit{Def}(x_k) \wedge m \in I$ **do**
    $\mathit{Val}(x_k) := \mathit{Val}(x_k) + a_m d$
    $\mathit{Def}(x_k) := a + \sum_{i \in I \setminus \{m\}} a_i x_i + a_m(c + \sum_{j \in J} c_j x_j)$
end

procedure Pivot
input
  x_b : basis variable
  act : activation direction
  x_i : undefined and active variable
begin
  $c + \sum_{j \in J} c_j x_j := \mathit{Def}(x_b)$
  $t := -\frac{1}{c_i}(c + \sum_{j \in J \setminus \{i\}} c_j x_j - x_b)$
  $\text{Substitute}(x_i, t)$
  $\mathit{Basis} := (\mathit{Basis} \setminus \{x_b\}) \cup \{x_i\}$
  $\text{Activate}(x_b, \mathit{act})$
  $\mathit{Active}(x_i) := \texttt{none}$
end

Figure 7.3: Algorithm in CLP(Q) page 1

```
procedure UpdateBound
input
  c + c_j x_j ≤ 0 : single variable linear inequality
begin
  if c_j > 0 then
    Status := UpdateUpper(x_j, −c/c_j)
    act := lower
  else
    Status := UpdateLower(x_j, −c/c_j)
    act := upper
  if Status = updated ∧ Def(x_j) ≠ x_j then
    if x_j ∉ Basis then
      a + Σ_{i∈I} a_i x_i := Def(x_j)
      if ∃k ∈ I Low(x_k) = −∞ ∧ Up(x_k) = +∞ then
        Substitute(x_k, −(1/a_k)(a + Σ_{i∈J\{k}} a_i x_i − x_j))
      else
        AddBasis(x_j, act)
    if x_j ∈ Basis then
      RepairVar(x_j)
end
```

```
procedure UpdateLower
input
  x_j : variable
  lb : Q
begin
  if Up(x_j) < lb then
   throw "Unsatisfiable"
  elsif Up(x_j) = lb then
   EnQueue( x_j = lb )
  else Low(x_j) < lb then
   if Active(x_j) = lower then
     _ := PushUp(x_j)
   Low(x_j) := lb
   if Active(x_j) = lower then
     Activate(x_j, lower)
   return updated
  return noChange
end
```

```
procedure UpdateUpper
input
  x_j : variable
  ub : Q
begin
  if Low(x_j) > ub then
   throw "Unsatisfiable"
  elsif Low(x_j) = ub then
   EnQueue( x_j = ub )
  else Up(x_j) > ub then
   if Active(x_j) = upper then
     _ := PushLow(x_j)
   Up(x_j) := ub
   if Active(x_j) = upper then
     Activate(x_j, upper)
   return updated
  return noChange
end
```

```
procedure AddBasis
input
  x_m : variable entering in basis
  act : preferred activation
begin
  Basis := Basis ∪ {x_m}
  Active(x_m) := none
  c + Σ_{j∈J} c_j x_j := Def(x_m)
  for each j ∈ J : Active(x_j) = none
  do
    if Low(x_j) = −∞ then
      Activate(x_j, upper)
    elsif Up(x_j) = +∞ then
      Activate(x_j, lower)
    elsif act = upper ⊕ c_j > 0 then
      Activate(x_j, lower)
    else
      Activate(x_j, upper)
end
```

```
procedure Activate
input
  x_m : variable
  act : activation direction
begin
  Active(x_m) := act
  match act with
   | lower -> new := Low(x_m)
   | upper -> new := Up(x_m)
  d := new − Val(x_m)
  for each x_k ∈ X :
  c + Σ_{i∈I} c_i x_i = Def(x_k) ∧ m ∈ I do
    Val(x_k) := Val(x_k) + c_m d
end
```

```
procedure EnQueue
input
  x_k = c : variable equality
begin
  queue := queue ∪ {x_k − c = 0}
end
```

Figure 7.4: Algorithm in CLP(Q) page 2

```
procedure RepairBasis
begin
  LocalBasis := Basis
  for each x_b ∈ LocalBasis ∧ x_b ∈ Basis do
     RepairVar(x_b)
end
```

```
procedure RepairVar
input
  x_b : basis variable
begin
  if Val(x_b) ≥ Up(x_b) then   RepairUp(x_b)
  if Val(x_b) ≤ Low(x_b) then   RepairLow(x_b)
end
```

```
procedure RepairUp
input
  x_b : basis variable
begin
  if Val(x_b) < Up(x_b) then   return
  c + Σ_{i∈I} c_i x_i := Def(x_b)
  if ∃i ∈ I. c_i > 0 ∧ Active(x_i) = upper then
     Status := PushLow(x_i)
  elsif ∃i ∈ I. c_i < 0 ∧ Active(x_i) = lower then
     Status := PushUp(x_i)
  else
     Status := optimum
     if Val(x_b) = Up(x_b) then
        EnQueue( x_b = Up(x_b) )
     else
        throw "Unsatisfiable"
  match Status with
   | applied -> RepairUp(x_b)
   | nobound(x_i) -> Pivot(x_b, upper, x_i)
end
```

```
procedure RepairLow
input
  x_b : basis variable
begin
  if Val(x_b) > Low(x_b) then   return
  c + Σ_{i∈I} c_i x_i := Def(x_b)
  if ∃i ∈ I. c_i > 0 ∧ Active(x_i) = lower then
     Status := PushUp(x_i)
  elsif ∃i ∈ I. c_i < 0 ∧ Active(x_i) = upper then
     Status := PushLow(x_i)
  else
     Status := optimum
     if Val(x_b) = Low(x_b) then
        EnQueue( x_b = Low(x_b) )
     else
        throw "Unsatisfiable"
  match Status with
   | applied -> RepairLow(x_b)
   | nobound(x_i) -> Pivot(x_b, lower, x_i)
end
```

```
procedure PushLow
input
  x_i : undefined and active variable
begin
  (lb, k) := (Low(x_i) − Up(x_i), i)
  for each x_b ∈ Basis :        (* Pick x_b in order *)
  Def(x_b) = c + Σ_{j∈J} c_j x_j ∧ i ∈ J do
     if c_i > 0 ∧ (Low(x_b) − Val(x_b))/c_i > lb then
        (lb, k, act) := ((Low(x_b) − Val(x_b))/c_i, b, lower)
     if c_i < 0 ∧ (Up(x_b) − Val(x_b))/c_i > lb then
        (lb, k, act) := ((Up(x_b) − Val(x_b))/c_i, b, upper)
  done
  if k = i ∧ Low(x_i) = −∞ then
     return nobound(x_i)
  elsif k = i then
     Activate(x_i, lower)
  else
     Pivot(x_k, act, x_i)
  return applied
end
```

```
procedure PushUp
input
  x_i : undefined and active variable
begin
  (ub, k) := (Up(x_i) − Low(x_i), i)
  for each x_b ∈ Basis :        (* Pick x_b in order *)
  Def(x_b) = c + Σ_{j∈J} c_j x_j ∧ i ∈ J do
     if c_i > 0 ∧ (Up(x_b) − Val(x_b))/c_i < ub then
        (ub, k, act) := ((Up(x_b) − Val(x_b))/c_i, b, upper)
     if c_i < 0 ∧ (Low(x_b) − Val(x_b))/c_i < ub then
        (ub, k, act) := ((Low(x_b) − Val(x_b))/c_i, b, lower)
  done
  if k = i ∧ Up(x_i) = +∞ then
     return nobound(x_i)
  elsif k = i then
     Activate(x_i, upper)
  else
     Pivot(x_k, act, x_i)
  return applied
end
```

Figure 7.5: Algorithm in CLP(Q) page 3

$t = 0$ is unsatisfiable, and we throw an exception "`Unsatisfiable`". If the de-referenced term contains an undefined and unbounded variable then this variable is substituted by the rest of de-referenced term in the solved form at line 7 and we get a solved form. Otherwise, we pick a variable appearing in the de-referenced term and the variable is substituted by the rest of de-referenced term in the solved form. Since, the variable is bounded and defined, we add the variable in basis at line 11. Note that we pass `lower` as the activation direction in the second argument of AddBasis, which is an arbitrary choice. These modifications of the solved form may leads to violation of condition (9), which is fixed by calling RepairBasis. The code after line 13 is already discussed in description of the global data structures.

Procedure AddInequality This procedure takes a linear inequality $t \leq 0$ as input. At line 1, t is de-referenced using the solved form. If the solved form implies $t \leq 0$ then the condition at line 2 is true and procedure continues at line 21. If the condition at line 4 is true then the conjunction of the solved form and $t \leq 0$ is unsatisfiable, and we throw an exception "`Unsatisfiable`". If either the input term or de-referenced term contains a single variable then UpdateBound is called at line 7 or 9, respectively. Otherwise, we introduce a slack variable x_k and initialize the lower bound of x_k with 0 at lines 11–13. Now we need to add an equality between x_k and the negation of the de-referenced term in the solved form. If the de-referenced term contains an undefined and unbounded variable then this variable is substituted by the rest of de-referenced term added with x_k in the solved form at line 15 and we get a solved form. Otherwise, AddInequality sets definition of x_k to negation of the de-referenced term and add x_k to the basis at line 17 and 19. Only x_k can violate condition (9). So RepairVar is called to fix the violation at line 20. The code after line 21 is already discussed in description of the global data structures.

Procedures RepairBasis and RepairVar RepairBasis iteratively changes the basis by pivot operations until condition (9) is satisfied. RepairBasis keeps a local copy of the current basis and then iterate over the variables that will remain in the basis after the call to RepairVar in each iteration. RepairVar takes a basis variable x_b as input, checks if $Val(x_b)$ violates any of its bounds, and calls accordingly RepairUp or RepairLow, accordingly.

Procedures RepairUp, RepairLow, PushLow, and PushUp We only discuss RepairUp and PushLow. The descriptions of RepairLow and PushUp are similar, respectively.

RepairUp takes a basis variable x_b as input. RepairUp recursively attempts to decrease $Val(x_b)$ such that $Val(x_b) < Up(x_b)$ or moves x_b out of the basis. The condition (8) defines $Val(x_b)$ in terms of values of variables appearing in $Def(x_b)$. The condition at line 3 holds if a variable x_i appears in $Def(x_b)$ with a positive coefficient and is activated with the direction `upper`. We can decrease $Val(x_b)$ by decreasing $Val(x_i)$. Since $Val(x_i)$ is taking the maximum allowed value, it can be decreased. At line 4, procedure PushLow is called to decrease value of $Val(x_i)$. The code at line 5 and 6 is symmetric therefore we will not discuss it. If both conditions at line 3 and 5 fail then we can not decrease $Val(x_i)$ any further, and the execution continues at line 8. If $Val(x_b)$ is equal to $Up(x_b)$ then we have detected an equality and this equality is pushed into *queue*. Otherwise, the solved form is unsatisfiable and exception "`Unsatisfiable`" is thrown. The return value of the call to PushLow at line 4 is stored in *Status*. *Status* equals to `applied` indicates that a progress in decreasing $Val(x_b)$ has been made. Then, we decrease $Val(x_b)$ further. *Status* equals to `nobound`(x_i) indicates that x_i can be decreased without any bound and by doing a pivot operation between x_b and x_i we can satisfy the conditions of the solved form.

In procedure PushLow at line 1, k is set equal to i and lb records the maximum change in value of $Val(x_i)$ allowed by $Low(x_i)$. Then at lines 2–8, PushLow iterates over the basis variables and finds a basis variable x_k that may impose maximum bound on smallest value of lb, i.e., change in value of $Val(x_i)$. There are three possible cases at lines 9–14. The first and second case occur when no bounding basis variable exists and k remains equal to i at line 9. The first case occurs if there is no lower bound of x_i. In this case, a value nobound(x_i) is returned indicating that $Val(x_i)$ can be decreased without any bound at line 10. The second case occurs if there is a lower bound on x_i. In this case, the activation direction of x_i is changed from `upper` to `lower` at line 12. This change leads to a decrease of $Val(x_b)$. The third case occurs if x_k is a basis variable. x_k is leaves the basis and x_i enters the basis at line 14. After the second and third cases, the execution continues at line 15 where `applied` is returned indicating to the caller that $Val(x_b)$ is decreased by some amount.

Procedures UPDATEBOUND, UPDATELOWER, and UPDATEUPPER The procedure UPDATEBOUND takes an inequality that contains only one variable as input and updates bounds of this variable. Depending on the variable coefficient in the input inequality, upper or lower bound is updated by calling UPDATEUPPER or UPDATELOWER, respectively.

We will discuss UPDATEUPPER. The description of UPDATELOWER is symmetric. UPDATEUPPER takes a variable x_j and a new upper bound ub for x_j as input. If ub is strictly lower than the lower bound of x_j then UPDATEUPPER throws an exception "`Unsatisfiable`" at line 2. If ub is equal to the lower bound of x_j then we have detected that x_j to be constant and the corresponding constant equality is stored in *queue* at line 4. If $Up(x_j) > ub > Low(x_j)$ then we update $Up(x_j)$. Due to conditions (7)–(9), updating an active bound is a difficult case. If $Active(x_j) = \texttt{upper}$ then PUSHLOW is called at line 7. If PUSHLOW moves x_j into the basis or changes its activation direction then the difficulty is eliminated. Otherwise, PUSHLOW makes no changes in solved form and solved form imposes no limit in decrease of upper bound. In both case, we update $Up(x_j)$ at line 8 without violating conditions (7)–(9) for any other variable. If $Active(x_j)$ is still equal to `upper` at line 9 then we update *Val* by calling ACTIVATE to satisfy condition (9). UPDATEUPPER returns value `updated` only upper bound is changed otherwise `noChange` is returned to the caller, i.e., UPDATEBOUND.

In UPDATEBOUND at line 7, if a bound of x_j is updated and x_j is a defined variable then lines 8–13 are executed to maintain condition (2) and (3). At line 14, if x_j is in the basis then we check and repair any violation of condition (9).

CLP(LI+UIF) using CLP(Q)

Figure 7.6 presents the CLP(LI+UIF) as an extension of CLP(Q). CLP(LI+UIF) solver extends CLP(Q) solver with a congruence checker for uninterpreted functions. The CLP(LI+UIF) contains an additional data structure *TermDef* that is a function from pairs of uninterpreted function symbols and lists of linear terms to a variable. *TermDef* is used to purify input atoms to produce linear atoms, and to check if a congruence axiom can be applied on input constraints and to produce new equalities. CLP(LI+UIF) takes $\mathcal{T}_{\mathrm{LI+UIF}}$ atoms as the input sequence. Given an input, the current solved form, the current *TermDef*, CLP(LI+UIF) computes the next solved form and *TermDef*. If CLP(LI+UIF) fails to compute the next solved form and *TermDef* then it throws an exception "`Unsatisfiable`". CLP(LI+UIF) adds the following three procedures.

Procedure ADDCONSTRAINT At line 1, PURIFY is called to remove uninterpreted functions from the input term and to produce a linear term. Next, the purified atom is added to CLP(Q) solver using its entry procedures at lines 2–4. If the call to an entry procedure of CLP(Q) does not throw an exception "`Unsatisfiable`" then CONGCHK is called at line 5 to check if congruence rules can be applied between any two of the terms stored in *TermDef*.

Procedure PURIFY This procedure takes a term in $\mathcal{T}_{\mathrm{LI+UIF}}$. PURIFY recursively traverses the input term in the bottom up order. During the traversal, PURIFY replaces each subterm whose top function symbol is uninterpreted with a variable. If the subterm is already seen before then the variable corresponding to the subterm is retrieved from *TermDef* at line 12. Otherwise, a fresh variable is chosen to replace for the subterm, and *TermDef* is updated accordingly at lines 9 and 10.

Procedure CONGCHK This procedure recursively executes until no new equality is detected from the solved form and *TermDef*. At lines 1–5, the new equalities are detected by the following if-condition. Let two variables x_j and x_k be in the range of *TermDef*. Assuming that in *TermDef*, x_j and x_k are mapped by the same function symbol f and lists of subterms $s_1, \ldots, s_m$ and $t_1, \ldots, t_m$, respectively. For all $i \in 1..m$, if the solved form implies $s_i = t_i$, which is checked by call to DEREF, then due to the congruence rule, $x_j = x_k$.

```
global variables
  TermDef ∅: Function symbols × linear term lists → X
```

```
procedure AddConstraint
input
  t ⋈ 0 : atom in T_LI+UIF
begin
  s := Purify(t)
  match ⋈ with
    | = -> AddEquality(s = 0)
    | ≤ -> AddInequality(s ≤ 0)
  CongChk()
end
```

```
procedure CongChk
begin
  if ∃x_j, x_k :
      x_j = TermDef(f, [t_1, ..., t_m]) ∧
      x_k = TermDef(f, [s_1, ..., s_m]) ∧
      ∀i ∈ 1..m. Deref(t_i − s_i) = 0 ∧
      Deref(x_j − x_k) ≠ 0
  then
      AddEquality(x_j − x_k = 0)
      CongChk()
end
```

```
procedure Purify
input
  t : term in T_LI+UIF
begin
  match t with
   | x_k -> return x_k
   | c t_1 -> return c Purify(t_1)
   | t_1 + t_2 -> return Purify(t_1) + Purify(t_2)
   | f(t_1, ..., t_m) ->
     for each i ∈ 1..m do
       s_i := Purify(t_i)
     if TermDef(f, [s_1, ..., s_m]) = ⊥ then
       pick fresh variable x_k
       TermDef(f, [s_1, ..., s_m]) := x_k
     else
       x_k := TermDef(f, [s_1, ..., s_m])
     return x_k
end
```

Figure 7.6: The extension for CLP(LI+UIF)

If $x_j = x_k$ is not already implied by the solved form, then we add this equality to the solved form. Note that equality detection algorithm is complete in our choice of solved form [53], therefore, CLP(LI+UIF) is complete as an unsatisfiability checker.

7.2 Cimmati et al.'s algorithm for proof production

In this section, we will first present an algorithm of Cimmati et al. [14], which we call CGS below, for extracting proof from an incremental simplex based satisfiablity checker. CGS is based on [23]. We will also show that this method can not be directly applied in the variant of incremental simplex algorithm used in our CLP(Q) solver.

CGS algorithm takes an unsatisfiable set of linear (in)equalities as input. This algorithm first pre-processes input constraints. Pre-processing involves two steps. First, equalities are removed from the input constraints. Each equality $t = 0$ is replaced with the conjunction of $0 \leq t$ and $t \leq 0$. Second, for each inequality $c + \sum_{j \in J} c_j x_j \leq 0$ with $|J| > 1$, a slack variable x_k is introduced[1] and $c + \sum_{j \in J} c_j x_j \leq 0$ is replaced with an equality $x_k = c + \sum_{j \in J} c_j x_j$ and inequality $x_k \leq 0$.

CGS algorithm runs the incremental simplex on these pre-processed constraints. We presented the solved form in previous section that is a variation of simplex tableau. We will use the notation of previous sections, but the conditions of solved form are not applicable in this section[2].

The initial simplex tableau is setup in the following way. *Def* of a slack variable is initialized using the equality generated due to its introduction and the original variables of input are initialized undefined. *Basis* is initialized with the set of slack variables. Variables are initialized as unbounded. We will use Def^0 to denote the initial value of *Def*.

[1] CGS [14] introduce slack variables even for single variable inequalities. This is not necessary for their algorithm.

[2] CGS uses a different version of simplex tableau as compare to CLP(Q) solved form. See [23] for details.

Now the rest of the constraints, which are only inequalities containing a single variable, are added to the simplex tableau one after another. The incremental simplex must terminate with unsatisfiable tableau because original constraints were unsatisfiable. The incremental simplex detects an unsatisfiablity by observing that a row in simplex tableau can not be made satisfiable by changing *Val*.

By analyzing the unsatisfiable row, a linear combination of input constraint that produces $1 \leq 0$ is derived. The pre-processing of CGS algorithm ensures that each variable appearing in the unsatisfiable row maps to an original inequality. Let

$$Def(x_k) = c + a_1 x_{k_1} + \cdots + a_i x_{k_i} + b_1 x_{k_{i+1}} + \cdots + b_j x_{k_{i+j}}$$

be the unsatisfiable row, where $a_1, \ldots, a_i < 0$ and $b_1, \ldots, b_j > 0$. We assume that the upper bound of x_k is violated. We consider the following proof of unsatisfiability of the input constraints.

$$\begin{bmatrix} 1 & -a_1 & \ldots & -a_i & b_1 & \ldots & b_j \end{bmatrix} \begin{bmatrix} Def^0(x_k) \leq Up(x_k) \\ Def^0(x_{k_1}) \leq Up(x_{k_1}) \\ \vdots \\ Def^0(x_{k_i}) \leq Up(x_{k_i}) \\ -Def^0(x_{k_{i+1}}) \leq -Low(x_{k_{i+1}}) \\ \vdots \\ -Def^0(x_{k_{i+j}}) \leq -Low(x_{k_{i+j}}) \end{bmatrix} = d \leq 0,$$

where $d > 0$. Inequalities appearing in the column vector must be in the input constraints or implied by one of the equalities in the input constraints. Note that *Up* or *Low* values are used from the unsatisfiable tableau. In the case of violation of the lower bound, *Up* and *Low* are interchanged.

See theorem 1 in [14] for the correctness of the above algorithm.

Example 4 (CGS algorithm)**.** We will apply CGS algorithm on the following unsatisfiable linear constraints.

$$x_1 + x_2 + 1 \leq 0 \wedge -x_1 + x_3 \leq 0 \wedge x_2 = 0 \wedge x_3 = 0$$

Pre-processing: Equalities are replaced with conjunctions of two linear inequalities as follows.

$$x_1 + x_2 + 1 \leq 0 \wedge -x_1 + x_3 \leq 0 \wedge \underbrace{x_2 \leq 0 \wedge x_2 \geq 0}_{x_2=0} \wedge \underbrace{x_3 \leq 0 \wedge x_3 \geq 0}_{x_3=0}$$

In our example, there are two linear inequalities where slack variables u and v are introduced. Following constraints are the result of introduction of slack variables.

$$\underbrace{u = x_1 + x_2 + 1 \wedge u \leq 0}_{x_1+x_2+1\leq 0} \wedge \underbrace{v = -x_1 + x_3 \wedge v \leq 0}_{-x_1+x_3\leq 0} \wedge x_2 \leq 0 \wedge x_2 \geq 0 \wedge x_3 \leq 0 \wedge x_3 \geq 0$$

Executing incremental Simplex: Slack variable equalities are used to initialize the tableau. The basis of the tableau is initialized with the set of slack variables. In Figure 7.7(a), at the top initialized simplex tableau is displayed and a subsequent execution of the incremental simplex is also presented. After adding each single variable linear inequality, the figure displays the resulting tableau. At the end, incremental simplex fails to find satisfiable tableau. The gray row corresponding to x_3 is responsible for failure. $Val(x_3) = -1$, which violates the lower bound on x_3 and *Val* of variables appearing in $Def(x_3)$ can not be changed in order to increase $Val(x_3)$.

Deriving unsatisfiablity proof: The unsatisfiability proof derived from the unsatisfiable row is

$$\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -Def^0(x_3) \leq -Low(x_3) \\ -Def^0(x_2) \leq -Low(x_2) \\ Def^0(u) \leq Up(u) \\ Def^0(v) \leq Up(v) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -x_3 \leq 0 \\ -x_2 \leq 0 \\ x_1 + x_2 + 1 \leq 0 \\ -x_1 + x_3 \leq 0 \end{bmatrix} = 1 \leq 0.$$

Initial tableau

	Def^0	Low	Up	Val
basis	$u = x_1 + x_2 + 1$	$-\infty$	$+\infty$	1
basis	$v = -x_1 + x_3$	$-\infty$	$+\infty$	0
	$x_1 = x_1$	$-\infty$	$+\infty$	0
	$x_2 = x_2$	$-\infty$	$+\infty$	0
	$x_3 = x_3$	$-\infty$	$+\infty$	0

$u \leq 0 \downarrow$ pivot(u,x_1)

Def	Low	Up	Val
$x_1 = u - x_2 - 1$	$-\infty$	$+\infty$	-1
$v = -u + x_2 + x_3 + 1$	$-\infty$	$+\infty$	1
$u = u$	$-\infty$	0	0
$x_2 = x_2$	$-\infty$	$+\infty$	0
$x_3 = x_3$	$-\infty$	$+\infty$	0

$v \leq 0 \downarrow$ pivot(v,x_3)

Def	Low	Up	Val
$x_1 = u - x_2 - 1$	$-\infty$	$+\infty$	-1
$x_3 = u + v - x_2 - 1$	$-\infty$	$+\infty$	-1
$v = v$	$-\infty$	0	0
$u = u$	$-\infty$	0	0
$x_2 = x_2$	$-\infty$	$+\infty$	0

$0 \leq x_2 \leq 0$, $x_3 \leq 0 \downarrow$ No pivot needed

Def	Low	Up	Val
$x_1 = u - x_2 - 1$	$-\infty$	$+\infty$	-1
$x_3 = u + v - x_2 - 1$	$-\infty$	0	-1
$v = v$	$-\infty$	0	0
$u = u$	$-\infty$	0	0
$x_2 = x_2$	0	0	0

$0 \leq x_3 \downarrow$ No pivot possible

Def	Low	Up	Val
$x_1 = u - x_2 - 1$	$-\infty$	$+\infty$	-1
$x_3 = u + v - x_2 - 1$	0	0	-1
$v = v$	$-\infty$	0	0
$u = u$	$-\infty$	0	0
$x_2 = x_2$	0	0	0

(a)

Initial tableau is empty

$x_1 + x_2 + 1 \leq 0 \downarrow$

Def	Low	Up	$Active$	Val
$x_2 = -x_1 - u - 1$	$-\infty$	$+\infty$	`none`	-1
$x_1 = x_1$	$-\infty$	$+\infty$	`none`	0
$u = u$	0	$+\infty$	`none`	0

$-x_1 + x_3 \leq 0 \downarrow$

Def	Low	Up	$Active$	Val
$x_2 = -x_1 - u - 1$	$-\infty$	$+\infty$	`none`	-1
$x_3 = x_1 - v$	$-\infty$	$+\infty$	`none`	-1
$x_1 = x_1$	$-\infty$	$+\infty$	`none`	0
$u = u$	0	$+\infty$	`none`	0
$v = v$	0	$+\infty$	`none`	0

$x_2 = 0 \downarrow$ SUBSTITUTE($x_1, -u - 1$)

Def	Low	Up	$Active$	Val
$x_3 = -u - v - 1$	$-\infty$	$+\infty$	`none`	-1
$x_1 = -u - 1$	$-\infty$	$+\infty$	`none`	0
$x_2 = 0$	$-\infty$	$+\infty$	`none`	0
$u = u$	0	$+\infty$	`none`	0
$v = v$	0	$+\infty$	`none`	0

$x_3 = 0 \downarrow$ SUBSTITUTE($u, -v - 1$)
ADDBASIS(u, `lower`)
No pivot possible

Def	Low	Up	$Active$	Val
$u = -v - 1$	0	$+\infty$	`none`	-1
$x_1 = v$	$-\infty$	$+\infty$	`none`	0
$x_3 = 0$	$-\infty$	$+\infty$	`none`	0
$x_2 = 0$	$-\infty$	$+\infty$	`none`	0
$v = v$	0	$+\infty$	`lower`	0

(b)

Figure 7.7: (a) Execution incremental simplex on the constraints obtained by pre-processing $x_1 + x_2 + 1 \leq 0 \land -x_1 + x_3 \leq 0 \land x_2 = 0 \land x_3 = 0$. (b) Execution of CLP(Q) on the same constraints.

Note that linear inequalities appearing in the proof are either appear in the input or implied by an equality in input constraints.

Why CGS algorithm cannot be used with our CLP(Q) solver? CGS algorithm does not require any modification in the incremental simplex. It pre-processes the input constraints in a way such that resulting unsatisfiable tableau directly represents the unsatisfiability proof. However, there is an implicit assumption in the algorithm. The algorithm assumes that if a variable is detected to have a constant value then incremental simplex must not propagate the constant value in the tableau. Equality propagation leads to the violation of the correctness of the algorithm. Our CLP(Q) solver does propagate equalities and moreover does not introduce slack variables for equalities, which are important optimizations of the solver. If we pre-process input constraints to split equalities into two inequalities and introduce slack variables then CLP(Q) will internally detect that the slack variables are equal to zero and they will be removed from the tableau.

In Figure 7.7(b), we show an execution of our CLP(Q) solver. At the last tableau, we find that only slack variables u and v are left in the unsatisfiable row. x_2 and x_3 are set to zero and propagated to the definitions of all other variables therefore the unsatisfiable row does not contain x_2 and x_3. Hence, we cannot construct unsatisfiability proof using unsatisfiable row anymore.

7.3 Our algorithm for proof production

In this section, we present our method of instrumenting CLP(LI+UIF) to extract unsatisfiability proofs. The instrumentation records how input facts are used to obtain the solved form. First we will discuss this idea in detail. Then, we will present the full instrumentation of CLP(LI+UIF). Finally for the efficiency of the instrumentation, we will discuss *lazy instrumentation* that evaluates the recorded information on demand.

Main idea

Each conjunct of Equation (7.1) is implied by the input constraints. Hence, each conjunct can be obtained by a linear combination of the input constraints. We record the linear combination that produces each linear inequality in the solved form. We introduce a *reason variable* corresponding to each input inequality. We call a linear term over these variables a *reason term*. A reason term represents a linear combination of the input inequalities. We also have equalities as input, which represent two inequalities. Therefore, for each equality we introduce a pair of reason terms.

We can rewrite Equation (7.1) as the following equation.

$$\bigwedge_{k=1}^{n} (\ 0 \leq \mathit{Def}(x_k) - x_k \leq 0 \wedge \mathit{Low}(x_k) \leq x_k \leq \mathit{Up}(x_k)\ )$$

Note that there are four linear inequalities for a variable x_k. In our instrumentation, we store a pair of reason terms $\Delta(x_k)$ that records derivation of $0 \leq \mathit{Def}(x_k) - x_k \leq 0$. We also store reason terms $\Delta_l(x_k)$ and $\Delta_u(x_k)$ that record derivations of $\mathit{Low}(x_k) \leq x_k \leq \mathit{Up}(x_k)$ respectively. We update these reason terms each time the solved form is updated.

Example 5 (Instrumentation)**.** Let us consider the input constraints in Example 4. We introduce reason variables α_1 for $x_1 + x_2 + 1 \leq 0$ and α_2 for $-x_1 + x_3 \leq 0$. We introduce pairs of reason terms $(-\alpha_3, \alpha_3)$ for $x_2 = 0$ and $(-\alpha_4, \alpha_4)$ for $x_3 = 0$. Note that α_3 represents $x_2 \leq 0$ and $-\alpha_3$ represents $-x_2 \leq 0$. After adding $x_1 + x_2 + 1 \leq 0$ and $-x_1 + x_3 \leq 0$ as input using instrumented CLP(Q), we obtain the following instrumented solved form.

Def	*Low*	*Up*	*Active*	*Val*	Δ	Δ_l	Δ_u
$x_2 = -x_1 - u - 1$	$-\infty$	$+\infty$	none	-1	$(0,0)$	0	0
$x_3 = x_1 - v$	$-\infty$	$+\infty$	none	-1	$(0,0)$	0	0
$x_1 = x_1$	$-\infty$	$+\infty$	none	0	$(0,0)$	0	0
$u = u$	0	$+\infty$	none	0	$(0,0)$	α_1	0
$v = v$	0	$+\infty$	none	0	$(0,0)$	α_2	0

In the above solved form, we introduced slack variables u and v and equalities $x_2 + x_1 + 1 + u = 0$ and $-x_1 + x_3 + v = 0$. These equalities are introduced by CLP(Q) internally therefore we do not assign a reason variable for them. The lower bounds of u and v are obtained by a linear combination of the input constraints and the above introduced equalities. $\Delta_l(u)$ and $\Delta_l(v)$ reflect only the contributions of the input constraints.[3]

After adding $x_2 = 0$, we obtain the following solved form.

Def	*Low*	*Up*	*Active*	*Val*	Δ	Δ_l	Δ_u
$x_3 = -u - v - 1$	$-\infty$	$+\infty$	none	-1	$(-\alpha_3, \alpha_3)$	0	0
$x_1 = -u - 1$	$-\infty$	$+\infty$	none	0	$(-\alpha_3, \alpha_3)$	0	0
$x_2 = 0$	$-\infty$	$+\infty$	none	0	$(\alpha_3, -\alpha_3)$	0	0
$u = u$	0	$+\infty$	none	0	$(0,0)$	α_1	0
$v = v$	0	$+\infty$	none	0	$(0,0)$	α_2	0

After adding $x_3 = 0$, we obtain the following solved form.

Def	*Low*	*Up*	*Active*	*Val*	Δ	Δ_l	Δ_u
$u = -v - 1$	**0**	$+\infty$	none	-1	$(-\boldsymbol{\alpha_3} - \boldsymbol{\alpha_4}, \alpha_3 + \alpha_4)$	$\boldsymbol{\alpha_1}$	0
$x_1 = v$	$-\infty$	$+\infty$	none	0	$(\alpha_4, -\alpha_4)$	0	0
$x_3 = 0$	$-\infty$	$+\infty$	none	0	$(\alpha_4, -\alpha_4)$	0	0
$x_2 = 0$	$-\infty$	$+\infty$	none	0	$(\alpha_3, -\alpha_3)$	0	0
$v = v$	**0**	$+\infty$	lower	0	$(0,0)$	$\boldsymbol{\alpha_2}$	0

The gray row in the above solved form is unsatisfiable. By analyzing the row, we conclude that reason term $\mathit{first}(\Delta(u)) + \Delta_l(u) + \Delta_l(v) = \alpha_1 + \alpha_2 - \alpha_3 - \alpha_4$ derives the unsatisfiability.

CLP(LI+UIF) with instrumentation

Figures 7.8, 7.9, and 7.10 present the instrumented CLP(Q) that implements the above idea of producing proofs. Figure 7.11 presents the instrumentation of CLP(LI+UIF) extension. The entry procedure is ProofGen that takes an unsatisfiable conjunction in $\mathcal{T}_{\text{LI+UIF}}$ as input and outputs a unsatisfiablity proof as a proof tree, which is introduced in Section 2.2. We have added * in the name of the original procedures to obtain name for instrumented version. The instrumented version of any procedure does all the operations as the original procedures along with the additional instrumentation code. We will only discuss the additional code. If we do not need to add any instrumentation in a procedure then it is not reproduced. If such procedures are called then the reader must refer to the presentation of CLP(LI+UIF) in Section 7.1.

Global data structures In instrumented CLP(Q), the global data structures also includes Δ, Δ_l, and Δ_u as defined above. In the instrumented CLP(LI+UIF) extension, the additional global data structures are a set of reason variables Υ, a map from Υ to the corresponding inequality Π, and a proof tree P. All these additional data structures are initialized to be empty.

The reason variables are introduced in CLP(LI+UIF) extension. The second parameters of entry procedures of instrumented CLP(Q) are the reason terms that derives the linear atoms passed passes as a first parameters.

[3]We do not record contribution of equalities introduced for slack variables because we only want to output proof for inequalities that do not contain slack variables.

global variables

$$\left.\begin{cases} X = \emptyset & : \text{set of variables} & \quad Basis = \emptyset & : \text{set of variables} \\ Def = \emptyset & : X \to \text{linear terms} & & \\ Low = \emptyset & : X \to \mathbb{Q} \cup \{-\infty\} & \quad Up = \emptyset & : X \to \mathbb{Q} \cup \{+\infty\} \\ Active = \emptyset & : X \to \{\texttt{none}, \texttt{lower}, \texttt{upper}\} & \quad Val = \emptyset & : X \to \mathbb{Q} \end{cases}\right\} : \text{solved form}$$

$queue = \emptyset$: set of linear constraints

$$\left.\begin{cases} \Delta = \emptyset & : X \to \text{reason term pair} & & \\ \Delta_l = \emptyset & : X \to \text{reason term} & \quad \Delta_u = \emptyset & : X \to \text{reason term} \end{cases}\right\} : \text{instrumentation}$$

---

procedure ADDEQUALITY*
input
  $t = 0$: linear constraint
  δ^p : reason term pair
begin
  (δ_l, δ_u) := DEREASON(t, δ^p)
  $c + \sum_{j \in J} c_j x_j$:= DEREF(t)
  if $c = 0 \wedge J = \emptyset$ **then**
    skip
  elsif $c < 0 \wedge J = \emptyset$ **then**
    throw "$\texttt{Proof}(\delta_l / - c)$"
  elsif $c > 0 \wedge J = \emptyset$ **then**
    throw "$\texttt{Proof}(\delta_u / c)$"
  elsif $\exists i \in J.\ Low(x_i) = -\infty \wedge Up(x_i) = +\infty$ **then**
    δ_i^p := SCALEREASONPAIR$((\delta_l, \delta_u), -\frac{1}{c_i})$
    SUBSTITUTE*$(x_i, -\frac{c + \sum_{j \in J \setminus \{i\}} c_j x_j}{c_i}, \delta_i^p)$
  else
    pick $i \in J$
    δ_i^p := SCALEREASONPAIR$((\delta_l, \delta_u), -\frac{1}{c_i})$
    SUBSTITUTE*$(x_i, -\frac{c + \sum_{j \in J \setminus \{i\}} c_j x_j}{c_i}, \delta_i^p)$
    ADDBASIS$(x_i, \texttt{lower})$
    REPAIRBASIS()
  if $(s = 0, \delta_s^p) \in queue$ **then**
    $queue := queue \setminus \{(s = 0, \delta_s^p)\}$
    ADDEQUALITY*$(s = 0, \delta_s^p)$
end

procedure ADDINEQUALITY*
input
  $t \leq 0$: linear constraint
  δ : reason term
begin
  $(\_, \delta_u)$:= DEREASON$(t, (0, \delta))$
  $c + \sum_{j \in J} c_j x_j$:= DEREF(t)
  if $c \leq 0 \wedge J = \emptyset$ **then**
    skip
  elsif $c > 0 \wedge J = \emptyset$ **then**
    throw "$\texttt{Proof}(\delta_u / c)$"
  elsif $t = a + a_i x_i$ **then**
    UPDATEBOUND*$(a + a_i x_i \leq 0, \delta)$
  elsif $J = \{j\}$ **then**
    UPDATEBOUND*$(c + c_j x_j \leq 0, \delta_u)$
  else
    pick fresh x_k (* slack variable *)
    INITIALIZE*(x_k)
    $Low(x_k)$:= 0
    $\Delta_l(x_k)$:= δ_u
    if $\exists i \in J. Low(x_i) = -\infty \wedge Up(x_i) = +\infty$ **then**
      SUBSTITUTE*$(x_i, -\frac{c + \sum_{j \in J \setminus \{i\}} c_j x_j + x_k}{c_i}, (0, 0))$
    else
      $Def(x_k)$:= $-(c + \sum_{j \in J} c_j x_j)$
      $Val(x_k)$:= $-(c + \sum_{j \in J} c_j\, Val(x_j))$
      ADDBASIS$(x_k, \texttt{lower})$
      REPAIRVAR(x_k)
  if $(s = 0, \delta_s^p) \in queue$ **then**
    $queue := queue \setminus \{(s = 0, \delta_s^p)\}$
    ADDEQUALITY*$(s = 0, \delta_s^p)$
end

---

procedure INITIALIZE*
input
  x_i : fresh variable
begin
  if $Def(x_i) = \perp$ **then**
  INITIALIZE(x_j)
  $\Delta(x_i)$:= $(0, 0)$
  $\Delta_l(x_i)$:= 0
  $\Delta_u(x_i)$:= 0
end

procedure SUBSTITUTE*
input
  x_m : variable
  $c + \sum_{j \in J} c_j x_j$: linear term
  δ^p : reason term pair
begin
  SUBSTITUTE$(x_m, c + \sum_{j \in J} c_j x_j)$
  REASONSUBSTITUTE(x_m, δ^p)
end

procedure PIVOT*
input
  x_b : basis variable
  act : activation direction
  x_i : undefined and active variable
begin
  PIVOT(x_b, act, x_i)
  $c + \sum_{j \in J} c_j x_j$:= $Def(x_b)$
  δ^p := SCALEREASONPAIR$(\Delta(x_b), -\frac{1}{c_i})$
  REASONSUBSTITUTE(x_l, δ^p)
end

Figure 7.8: Instrumented version of CLP(Q)

procedure UpdateBound*
input
$c + c_j x_j \leq 0$: single variable linear inequality
δ : reason term
begin
if $c_j > 0$ **then**
$Status$:= UpdateUpper*$(x_j, -c/c_j, \delta/c_j)$
act := `lower`
else
$Status$:= UpdateLower*$(x_j, -c/c_j, -\delta/c_j)$
act := `upper`
if $Status = \texttt{updated} \wedge Def(x_j) \neq x_j$ **then**
if $x_j \notin Basis$ **then**
$a + \sum_{i \in I} a_i x_i$:= $Def(x_j)$
if $\exists k \in I\ Low(x_k) = -\infty \wedge Up(x_k) = +\infty$ **then**
δ^p := ScaleReasonPair$(\Delta(x_j), -\frac{1}{a_k})$
Substitute*$(x_k, -\frac{1}{a_k}(a + \sum_{i \in J \setminus \{k\}} a_i x_i - x_j), \delta^p)$
else
AddBasis(x_j, act)
if $x_j \in Basis$ **then**
RepairVar(x_j)
end

procedure UpdateLower*
input
x_j : variable
lb : $\mathbb{Q}$
δ : reason term
begin
if $Up(x_j) < lb$ **then**
throw "`Proof`$(\frac{\delta + \Delta_u(x_j)}{lb - Up(x_j)})$"
elsif $Up(x_j) = lb$ **then**
EnQueue*($x_j = lb$,
$(\delta, \Delta_u(x_j))$)
else $Low(x_j) < lb$ **then**
if $Active(x_j) = \texttt{lower}$ **then**
_ := PushUp(x_j)
$Low(x_j)$:= lb
if $Active(x_j) = \texttt{lower}$ **then**
Activate$(x_j, \texttt{lower})$
return `updated`
return `noChange`
end

procedure UpdateUpper*
input
x_j : variable
ub : $\mathbb{Q}$
δ : reason term
begin
if $Low(x_j) > ub$ **then**
throw "`Proof`$(\frac{\delta + \Delta_l(x_j)}{Low(x_j) - ub})$"
elsif $Low(x_j) = ub$ **then**
EnQueue*($x_j = ub$,
$(\Delta_l(x_j), \delta)$)
else $Up(x_j) > ub$ **then**
if $Active(x_j) = \texttt{upper}$ **then**
_ := PushLow(x_j)
$Up(x_j)$:= ub
if $Active(x_j) = \texttt{upper}$ **then**
Activate$(x_j, \texttt{upper})$
return `updated`
return `noChange`
end

procedure EnQueue*
input
$x_k = c$: variable equality
δ^p : reason term pair
begin $queue$:= $queue \cup \{(x_k - c = 0, \delta^p)\}$ **end**

procedure ScaleReasonPair
input
(δ_l, δ_u) : reason term pair
λ : $\mathbb{Q}$
begin
if $\lambda < 0$ **then** (δ_l, δ_u) := (δ_u, δ_l)
return $(|\lambda|\delta_l, |\lambda|\delta_u)$
end

procedure ReasonSubstitute
input
x_j : variable
δ^p : reason term pair
begin
for each $x_k \in X$:
$c + \sum_{i \in I} c_i x_i = Def(x_k) \wedge j \in I$ **do**
(δ_l, δ_u) :=
ScaleReasonPair(δ^p, c_j)
(μ_l, μ_u) := $\Delta(x_k)$
$\Delta(x_k)$:= $(\delta_l + \mu_l, \delta_u + \mu_u)$
end

procedure DeReason
input
$c + \sum_{j \in J} c_j x_j$: linear term
(δ_l, δ_u) : initial reason term pair
begin
for each $j \in J$ **do**
Initialize*(x_j)
(μ_l, μ_u) :=
ScaleReasonPair$(\Delta(x_j), c_j)$
$(\delta_l, \delta_u) := (\delta_l + \mu_l, \delta_u + \mu_u)$
return (δ_l, δ_u)
end

procedure OptimaReason
input
x_b : basis variable
act : bounding direction
begin
$c + \sum_{i \in I} c_i x_i$:= $Def(x_b)$
match act **with**
| `lower ->` $(\delta, \_)$:= $\Delta(x_b)$
| `upper ->` $(\_, \delta)$:= $\Delta(x_b)$
for each $i \in I$ **do**
if $act = \texttt{lower} \oplus c_i < 0$ **then**
δ := $\delta + \Delta_l(x_i)$
then
δ := $\delta + \Delta_u(x_i)$
return δ
end

Figure 7.9: Instrumented version of CLP(Q) page 2

```
procedure RepairBasis*
begin
  LocalBasis := Basis
  for each x_b ∈ LocalBasis ∧ x_b ∈ Basis do
     RepairVar*(x_b)
end
```

```
procedure RepairVar*
input
  x_b : basis variable
begin
  if Val(x_b) ≥ Up(x_b) then RepairUp*(x_b)
  if Val(x_b) ≤ Low(x_b) then RepairLow*(x_b)
end
```

```
procedure RepairUp*
input
  x_b : basis variable
begin
  if Val(x_b) < Up(x_b) then   return
  c + Σ_{i∈I} c_i x_i := Def(x_b)
  if ∃i ∈ I. c_i > 0 ∧ Active(x_i) = upper then
     Status := PushLow*(x_i)
  elsif ∃i ∈ I. c_i < 0 ∧ Active(x_i) = lower then
     Status := PushUp*(x_i)
  else
     Status := optimum
     δ := OptimaReason(x_b, lower)
     if Val(x_b) = Up(x_b) then
       EnQueue*( x_b = Up( x_b), (δ, Δ_u(x_b)) )
     Else
       throw "Proof( (δ+Δ_u(x_b)) / (Val(x_b)−Up(x_b)) )"
  match Status with
   | applied -> RepairUp(x_b)
   | nobound(x_i) -> Pivot(x_b, upper, x_i)
end
```

```
procedure RepairLow*
input
  x_b : basis variable
begin
  if Val(x_b) > Low(x_b) then   return
  c + Σ_{i∈I} c_i x_i := Def(x_b)
  if ∃i ∈ I. c_i > 0 ∧ Active(x_i) = lower then
     Status := PushUp*(x_i)
  elsif ∃i ∈ I. c_i < 0 ∧ Active(x_i) = upper then
     Status := PushLow*(x_i)
  else
     Status := optimum
     δ := OptimaReason(x_b, upper)
     if Val(x_b) = Low(x_b) then
       EnQueue*( x_b = Low(x_b), (Δ_l(x_b), δ) )
     Else
       throw "Proof( (δ+Δ_l(x_b)) / (Low(x_b)−Val(x_b)) )"
  match Status with
   | applied -> RepairLow(x_b)
   | nobound(x_i) -> Pivot(x_b, lower, x_i)
end
```

```
procedure PushLow*
input
  x_i : undefined and active variable
begin
  (lb, k) := (Low(x_i) − Up(x_i), i)
  for each x_b ∈ Basis :      (* Pick x_b in order *)
  Def(x_b) = c + Σ_{j∈J} c_j x_j ∧ i ∈ J do
     if c_i > 0 ∧ (Low(x_b)−Val(x_b))/c_i > lb then
       (lb, k, act) := ((Low(x_b)−Val(x_b))/c_i, b, lower)
     if c_i < 0 ∧ (Up(x_b)−Val(x_b))/c_i > lb then
       (lb, k, act) := ((Up(x_b)−Val(x_b))/c_i, b, upper)
  done
  if k = i ∧ Low(x_i) = −∞ then
     return nobound(x_i)
  elsif k = i then
     Activate(x_i, lower)
  else
     Pivot*(x_k, act, x_i)
  return applied
end
```

```
procedure PushUp*
input
  x_i : undefined and active variable
begin
  (ub, k) := (Up(x_i) − Low(x_i), i)
  for each x_b ∈ Basis :      (* Pick x_b in order *)
  Def(x_b) = c + Σ_{j∈J} c_j x_j ∧ i ∈ J do
     if c_i > 0 ∧ (Up(x_b)−Val(x_b))/c_i < ub then
       (ub, k, act) := ((Up(x_b)−Val(x_b))/c_i, b, upper)
     if c_i < 0 ∧ (Low(x_b)−Val(x_b))/c_i < ub then
       (ub, k, act) := ((Low(x_b)−Val(x_b))/c_i, b, lower)
  done
  if k = i ∧ Up(x_i) = +∞ then
     return nobound(x_i)
  elsif k = i then
     Activate(x_i, upper)
  else
     Pivot*(x_k, act, x_i)
  return applied
end
```

Figure 7.10: Algorithm in instrumented CLP(Q) page 3

global variables

$TermDef := \emptyset$: Function symbols $\times$ linear term lists $\rightarrow X$

$$\left.\begin{cases} \Upsilon := \emptyset & : \text{ set of reason variables} \\ \Pi := \emptyset & : \Upsilon \rightarrow \text{inequalities} \\ P := \emptyset & : \text{atoms} \times \text{labels} \times \text{atoms}^* \end{cases}\right\} : \text{instrumentation}$$

```
procedure ProofGen
input
  Γ : conjunction of atoms in T_{LI+UIF}
begin
  try
    for each t ⋈ 0 from Γ do
      AddConstraint*(t ⋈ 0)
  catch "Proof(δ)"
    return P ∪ ReasonComb(1 ≤ 0, δ)
end
```

```
procedure AddConstraint*
input
  t ⋈ 0 : atom in T_{LI+UIF}
begin
  pick fresh reason variable α
  Υ := Υ ∪ {α}
  Π(α) := t ≤ 0
  s := Purify(t)
  P := P ∪ (t ≤ 0, Hyp, ())
  match ⋈ with
   | = ->
     P := P ∪ (−t ≤ 0, Hyp, ())
     AddEquality*(s = 0, (−α, α))
   | ≤ ->
     AddInequality*(s ≤ 0, α)
  CongChk*()
end
```

```
procedure ReasonComb
input
  t ≤ 0 : inequality
  λ_1 α_1 + ··· + λ_n α_n : reason term
begin
  for each i ∈ 1..n do
    t_i ≤ 0 := Π(α_i)
    if λ_i < 0 then (λ_i, t_i) := (−λ_i, −t_i)
  return ( t ≤ 0, PComb(λ_1, ..., λ_n),
                       (t_1 ≤ 0, ..., t_n ≤ 0) )
end
```

```
procedure CongChk*
begin
  if ∃x_j, x_k :
     x_j = TermDef(f, [t_1, ..., t_m]) ∧
     x_k = TermDef(f, [s_1, ..., s_m]) ∧
     ∀i ∈ 1..m. Deref(t_i − s_i) = 0 ∧
     Deref(x_j − x_k) ≠ 0
  then
     for each i ∈ 1..m do
       (δ_l, δ_u) := DeReason(t_i − s_i, (0, 0))
       p_i := DePurify(t_i − s_i)
       P := P ∪ ReasonComb(p_i ≤ 0, δ_u)
               ∪ ReasonComb(−p_i ≤ 0, δ_l)
     t := DePurify(x_j − x_k)
     P := P ∪ ( t ≤ 0, PCong,
                (p_1 ≤ 0, −p_1 ≤ 0, ..., p_n ≤ 0, −p_n ≤ 0) )
              ∪ ( −t ≤ 0, PCong,
                (−p_1 ≤ 0, p_1 ≤ 0, ..., −p_n ≤ 0, p_n ≤ 0) )
     pick fresh reason variable α
     Υ := Υ ∪ {α}
     Π(α) := t ≤ 0
     AddEquality*(x_j − x_k = 0, (−α, α))
     CongChk*()
end
```

```
procedure DePurify
input
  c + Σ_{i∈I} c_i x_i : linear term
begin
  t := c
  for each i ∈ I do
    if x_i = TermDef(f, [t_1, ..., t_m]) then
      for each j ∈ 1..m do
        s_j := DePurify(t_j)
      t := t + c_i f(s_1, ..., s_m)
    else
      t := t + c_i x_i
end
```

Figure 7.11: Instrumented version of CLP(LI+UIF)

Procedures Initialize*, Substitute*, Pivot*, and EnQueue* These procedures also update instrumented data structures along with the actions performed by the corresponding original procedures. If a variable is not yet part of the solved form then Initialize* calls Initialize to add the variable in the solved form and initialize all the reason terms corresponding to the variables to 0. Substitute* takes three parameters. The first two parameters are used to call Substitute, which does the substitution. The last one is a reason term pair δ^p that encodes the linear combinations of input inequalities that derives the substitution term. ReasonSubstitute is called to update Δ to reflect of the changes in the solved form. Pivot* calls Pivot along with the same input parameters and then updates Δ to reflect of the changes in the solved form by calling ReasonSubstitute at lines 2–4. EnQueue* takes as input a reason term pair along with the linear equality to be added into solved form. Now *queue* stores a pairs of linear equalities and reason term pairs.

Procedures ScaleReasonPair, ReasonSubstitute, DeReason, and OptimaReason These procedures are added to instrumented CLP(Q) to process reason terms. ScaleReasonPair takes a reason term pair (δ_l, δ_u) and a rational number λ and returns the scaler product of λ and the reason term pair. If λ is negative then δ_l and δ_u exchange their places and are scaled by the absolute value of λ.

ReasonSubstitute is called each time a variable x_m is substituted with a term. The reason term pair δ^p that derives the equality between x_m and the substituted term is also passed as the second parameter to ReasonSubstitute. This procedure iterate over all variables of the solved form and updates Δ to reflect changes in the solved form.

DeReason is called along with Deref by the entry procedures of instrumented CLP(Q). Deref returns a term that is less than or equal to zero and is implied by input atom to the entry procedures and the solved form. DeReason returns a reason term pair that derives the (in)equality returned by Deref.

If RepairUp* or RepairLow* finds that a bound is imposed on a basis variable x_b by the variables appearing in its definition then OptimaReason is called to obtain the reason term that derives the bound.

Procedures AddEquality* and AddInequality* These procedures are entry procedures of the instrumented CLP(Q). At the tail of the entry procedures the code for adding equalities into solved form that are stored in *queue* is modified to also pass the reason term pairs that derives the equalities to AddEquality* at lines 20 and 25 respectively.

AddEquality* takes a linear equality and a reason term pair as input. At line 1, DeReason is called to compute a reason term pair that derives dereferenced equality. If the dereferencing leads to unsatisfiability detection then at lines 6 or 8 an exception is thrown. This exception contains a reason term that derives $1 \leq 0$. At lines 11 and 15, a call to Substitute* is made. Just before these calls at lines 10 and 14, we compute the reason term pairs that derives the substitutions.

AddInequality* takes a linear inequality and a reason term as input. At line 1, DeReason is called to compute a reason term δ_u that derives the dereferenced inequality. DeReason returns a pair but we are only interested in the second component of the pair and the first component is not used. If the dereference inequality implies unsatisfiability then at lines 6 an exception containing a reason term that derives $1 \leq 0$ is thrown. At line 8 and 10, UpdateBound* is called with a linear inequality passed as first parameter and a reason term as the second parameter that derives the linear inequality. At line 12, a slack variable is introduced and we set reason of its lower bound as δ_u at line 15. We also introduced an equality between the slack variable and negation of dereferenced term and added to the solved form at lines 16–22. For this equality, we introduce the reason term pair $(0, 0)$ as discussed earlier.

Procedures UpdateBound*, UpdateLower*, and UpdateUpper* UpdateBound* takes an additional reason term as the second parameter along with the linear inequality that it derives. At lines 2 and 5, UpdateUpper* and UpdateLower* are called with additional third parameter that is a scaled value of the input reason term and derives the new bound. At line 12, Substitute* is called. So, we compute at line 11 the reason term pair that derives this substitution and pass it to the Substitute* as second parameter.

We will only discuss instrumentation in UPDATEUPPER*. The instrumentation in UPDATELOWER* is similar. UPDATEUPPER* takes a reason term that derives the new lower bound as an additional parameter. If the unsatisfiability at line 1 is true then an exception is thrown containing a reason term that derives that the new lower bound is greater than already existing upper bound in the solved form at line 2. If the new lower bound and the already existing upper bound are equal then an equality is added along with a reason term pair that derives the equality at line 4. The rest of the procedure is unmodified.

Procedures REPAIRBASIS*, REPAIRVAR*, REPAIRUP*, REPAIRLOW*, PUSHLOW*, and PUSHUP* There are no significant modifications in REPAIRBASIS*, REPAIRVAR* PUSHLOW*, and PUSHUP*. They are reproduced because they call the procedures that are modified.

We will only discuss instrumentation in REPAIRUP*. The instrumentation in REPAIRLOW* is similar. At line 9, OPTIMAREASON is called to compute the reason term that implies the lower bound on the input basis variable by the variables appearing in its definition. If the upper bound on the input basis variable is equal to this lower bound then an equality is added along with a reason term pair that derives the equality at line 11. Otherwise, an exception containing the reason term that derives $1 \leq 0$ is thrown.

Instrumentation of CLP(LI+UIF) **extension** PROOFGEN takes a conjunction of atoms and adds them into the solved form by calling ADDCONSTRAINT*. If the input conjunction is unsatisfiable, the one of the calls to ADDCONSTRAINT* throws an exception containing a reason term that derives $1 \leq 0$. This exception is caught at line 4 and a proof tree the proves unsatisfiability is returned at line 5.

REASONCOMB and DEPURIFY are additional procedures to support proof generation. REASONCOMB takes an inequality and a reason term that derives the inequality as input and returns an edge of the proof tree. DEPURIFY is an inverse of PURIFY. DEPURIFY takes a linear term as input and replaces variables appearing in the term with the corresponding term definitions from *TermDef*.

ADDCONSTRAINT* takes an atom $t \bowtie 0$ as input. ADDCONSTRAINT* introduces a fresh reason variable α, adds this fresh reason variable to Υ, and updates $\Pi(\alpha)$ with $t \leq 0$ at lines 1–3. At line 5, ADDCONSTRAINT* adds an edge in the proof tree expressing that $t \leq 0$ is derived by PHYP rule. If $\bowtie$ is an equality then ADDCONSTRAINT* also adds an edge in the proof tree expressing that $-t \leq 0$ is derived by PHYP rule at line 8. The calls to ADDEQUALITY* and ADDINEQUALITY* are instrumented with second parameters containing the reason terms that derive the atoms passed as first parameters.

If the condition for applying PCONG rule is true then at lines 7–16 CONGCHK* update the proof tree by recording the application of PCONG rule. In the loop at line 7, proof edges that derive antecedents of PCONG rule are added to the proof tree and then at line 13 the proof tree is added with the proof edges corresponding to application of the proof rule. Due to the application of PCONG rule, a fresh equality will be added to the solved form and we need to track its contributions in subsequent derivations. So CONGCHK* introduces a fresh reason variable and passes it to the ADDEQUALITY* at lines 17–20.

Lazy instrumentation

The instrumentation adds extra computation that may lead to significant increase in running time of CLP(LI+UIF). All operations of the instrumentation are done by SCALEREASONPAIR, REASONSUBSTITUTE, and DEREASON. These procedures can be implemented lazily since their results are not required for any decision in the instrumented CLP(LI+UIF). If an unsatisfiability is detected only then we may need to evaluate the results of these procedures. So, we can have a CLP(LI+UIF) that does not have addition running time and if we need a proof of the unsatisfiability only then we do extra work.

Chapter 8

Solving recursion-free Horn clauses over LI+UIF

Constraint solving is a vehicle of software verification that provides symbolic reasoning techniques for dealing with assertions describing program behaviors. In particular, abstraction and refinement techniques greatly benefits from applying constraint solving, where interpolation techniques [3,15,44,46,65,66] play a prominent role today.

Certain abstraction refinement tasks cannot be *directly* expressed as an interpolation question. For example, abstraction refinement for imperative programs with procedures [46], for higher order functional programs [57, 81], require additional pre-processing that splits discovered spurious counterexamples in multiple ways and applies interpolation on each splitting. Alternatively, as exemplified by an abstraction refinement procedure for multi-threaded programs [37], this preprocessing and series of interpolation computations can be expressed using a single constraint that consists of a finite set of recursion-free Horn clauses interpreted over the logical theory that is used to describe program behaviors.

In this chapter, we present an algorithm for solving Horn clauses over a combination of linear arithmetic, uninterpreted functions, and queries. Our algorithm opens new possibilities for the development of abstraction refinement schemes by providing the verification method designer an expressive, declarative way to specify what the refinement procedure needs to compute using Horn clauses. Several existing abstraction refinement schemes can directly benefit from our algorithm, e.g., for programs with procedures [44, 46], for multi-threaded programs [37], and for higher-order functional programs [57, 81, 83].

Technically, we present a generalization of partial interpolants, which are presented in chapter 6, to partial solutions for recursion-free Horn clauses, i.e., clauses that do not have cyclic dependencies between the occurring queries. Our algorithm follows a general scheme of combining interpolation procedures for different theories [85].

This chapter is organized as follows. Section 8.1 provides a formal definition of recursion-free Horn clauses and their solutions. We present the solving algorithm in Section 8.2 and discuss its correctness and complexity in Section 8.3. Section 8.4 illustrates how abstraction refinement tasks yield sets of Horn clauses and Section 8.5 illustrates how these sets of clauses are solved using our algorithm.

8.1 Recursion-free Horn clauses

We present auxiliary functions and recursion-free Horn clauses over linear arithmetic and uninterpreted functions. We use the notation for the theory of linear arithmetic and uninterpreted functions from Section 2.2.

Syntax

We assume countable sets of *variables* X, with $x \in X$, and *predicate symbols* $\mathcal{P}$, with $p \in \mathcal{P}$. Let the arity of predicate symbols be encoded in their names. Recall A is an atom in $\mathcal{T}_{\text{LI+UIF}}$. The following grammar defines Horn clauses.

$$\begin{array}{lll@{\qquad}lll} \text{queries} \ni Q & ::= & p(x, \ldots, x) & \text{heads} & \ni H & ::= A \mid Q \mid \mathit{false} \\ \text{bodies} \ni B & ::= & A \mid Q \mid B \wedge B & \text{Horn clauses} & \ni W & ::= B \rightarrow H \end{array}$$

Each Horn clause is implicitly universally quantified over the variables that appear in the clause.

A set of Horn clauses defines a binary *dependency* relation on predicate symbols. A predicate symbol $p \in \mathcal{P}$ *depends* on a predicate symbol $p_i \in \mathcal{P}$ if there is a Horn clause

$$\cdots \wedge p_i(\ldots) \wedge \cdots \rightarrow p(\ldots) \ ,$$

i.e., when p appears in the head of a clause that contains p_i in its body. A set of Horn clauses is *recursion-free* if the corresponding dependency relation does not contain any cycles. A set of Horn clauses is *tree-like* if 1) each predicate symbol appears at most once in the set of bodies and at most once in the set of heads of the given clauses, 2) there is no clause with an atom in its head, 3) there is one clause whose head is *false*.

For the rest of the chapter, we consider a finite set of Horn clauses $\mathcal{HC}$ that satisfies the following conditions. We assume that each variable occurs in at most one clause and that all variables occurring in each query are distinct. These assumptions simplify our presentation and can be established by an appropriate variable renaming and additional (in)equality constraints. Furthermore, we assume that $\mathcal{HC}$ is recursion-free and tree-like. The recursion-free assumption is critical for ensuring termination of the solving algorithm presented in this paper. The tree-like assumption simplifies our presentation without imposing any restrictions on the algorithm's applicability. Any finite set of recursion-free clauses can be transformed into the tree-like form. The solution for the computed tree-like form can be translated into the solution for the original set of clauses.

Auxiliary definitions

We assume the following standard functions. For dealing with trees, let $\mathit{nodes}(T)$ be the nodes of a tree T, $\mathit{root}(T)$ be the root node of T, $\mathit{leaves}(T)$ be the leaves of T, and $\mathit{subtree}(o, T)$ be the subtree of T rooted in its node o.

Let $\mathit{mgu}((Q_1, \ldots, Q_n), (Q'_1, \ldots, Q'_n))$ be the most general unifier between two sequences of queries if it exists, where a unifier is a solution to the conjunction of equations $Q_1 = Q'_1 \wedge \cdots \wedge Q_n = Q'_n$. We write $t\sigma$ for the application of a unifier σ on a term t, and we assume a canonical extension of the unifier application to constraints and their combination into sequences and sets.

Semantics

Let Γ be a function from queries to constraints. We assume that no two queries in the domain of Γ have an equal predicate symbol. We use this function to transform the set of Horn clauses containing queries into a set of query-free clauses as follows. In each clause $W \in \mathcal{HC}$ we replace each query Q in W with the constraint $\Gamma(Q')\sigma$ where Q' is in the domain of Γ, queries Q' and Q have an equal predicate symbol, and $\sigma = \mathit{mgu}(Q, Q')$. Let $\mathcal{HC}_\Gamma$ be the resulting set of clauses. Γ is a *solution* for $\mathcal{HC}$ if each clause in $\mathcal{HC}_\Gamma$ is a valid implication, and the following condition holds for the uninterpreted function symbols occurring in the range of the solution function. An uninterpreted function symbol f can occur in the solution $\Gamma(Q)$ for a query q if f appears in a Horn clause from $\mathcal{HC}$ whose head depends on Q and in a Horn clause from $\mathcal{HC}$, whose head does not depend on Q.

8.2 Algorithm

Our goal is an algorithm for computing solutions for recursion-free Horn clauses over linear arithmetic, uninterpreted functions, and queries. This section presents our solving algorithm HcSolve.

algorithm HcSolve
input
$\mathcal{HC}$: Horn clauses
vars
R : resolution tree
C : conjunctive constraint
P : proof tree
A : annotated proof tree
output
Γ : solution
begin
1 R := exhaustively apply RInit and RStep on $\mathcal{HC}$
2 $C := \bigwedge \mathit{leaves}(R)$
3 **if** exists P inferred from C such that P proves $\models C \rightarrow 1 \leq 0$ **then**
4 A := exhaustively apply HcHyp, HcComb, and HcCong on P
5 $\mathit{false}\ [\ \Pi\ ] := \mathit{root}(A)$
6 $\Gamma := \{(o, S) \mid (o, S) \in \Pi \wedge o \notin \mathit{leaves}(R) \cup \{\mathit{false}\}\}$
7 **return** Γ
8 **else**
9 **return** "no solution exists"
end.

Figure 8.1: Solving algorithm HcSolve. Line 5 extracts the partial solution Π annotating the root node of A. Line 6 obtains Γ by restricting the domain of Π to intermediate nodes of R, which are labeled by queries.

See Figure 8.1. The algorithm HcSolve consists of the following main steps. First, we compute a resolution tree R on the given set of Horn clauses. Next, we take a conjunction C of the leaves of the resolution tree and attempt to find a proof of its unsatisfiability. If no such proof can be found, then we report that there is no solution for the given set of Horn clauses. Otherwise, we proceed with the given proof by annotating its steps. Each intermediate atom occurring in proof tree is annotated by a function that assigns constraints to nodes of the resolution tree. Finally, the annotation of the root of the proof yields a solution for the given set of Horn clauses.

In the rest of this section we present the main steps of HcSolve.

Resolution tree

We put together individual Horn clauses from $\mathcal{HC}$ by applying resolution inference. A *resolution tree* keeps the intermediate results of this computation. An edge of a *resolution tree* is a sequence of queries and atoms that is terminated by a query or *false*. Each edge consists of $n > 2$ elements. The first $n-1$ elements represent the children nodes and the n-th element represents the parent node.

Given the set of Horn clauses $\mathcal{HC}$, we compute the corresponding resolution tree by applying the inference rules shown in Figure 8.2. Each rule takes as a premise a set of resolution trees together with a Horn clause and infers an extended resolution tree.

The rule RInit initiates the resolution tree computation by inferring a tree from each clause $A_1 \wedge \cdots \wedge A_m \rightarrow H$ that does not have any queries in its body. The atoms $A_1, \ldots, A_m$ become the children of the node H. The rule RStep extends a set of trees computed so far using a Horn clause. The extension is only possible if the root nodes of the respective trees can be unified with the queries occurring in the body of the clause. This restriction is formalized by the side condition requiring the existence of the most general unifier σ. The computed unifier is applied on the trees and the clause before they are combined into an extended resolution tree.

The resolution tree computation terminates since $\mathcal{HC}$ is recursion-free. Let R be the resulting tree. We

$$\text{RInit}\ \frac{A_1 \wedge \cdots \wedge A_m \to H}{\{(A_1, \ldots, A_m, H)\}}$$

$$\text{RStep}\ \frac{\begin{array}{c} R_1 \quad \ldots \quad R_n \\ Q_1 \wedge \cdots \wedge Q_n \wedge A_1 \wedge \cdots \wedge A_m \to H \end{array}}{\begin{array}{l} R_1\sigma \cup \cdots \cup R_n\sigma \cup \\ \{(Q_1, \ldots, Q_n, A_1, \ldots, A_m, H)\}\sigma \end{array}}\ \sigma = mgu((root(R_1), \quad \ldots, \quad root(R_n)), (Q_1, \ldots, Q_n))$$

Figure 8.2: Resolution tree inference rules RInit and RStep.

consider the set of leaves of the tree, and take their conjunction $C = \bigwedge leaves(R)$.

For a node o of the resolution tree, we define $InSmb(o)$ to be variables and uninterpreted function symbols that occur in atoms in the leaves of the subtree of o, and let $OutSmb(o)$ be variables and uninterpreted function symbols that occur in the leaves outside of the subtree of o. Formally, we have

$$\begin{aligned} InSmb(o) &= \bigcup\{Smb(o') \mid o' \in leaves(subtree(o, R))\}\ , \\ OutSmb(o) &= \bigcup\{Smb(o') \mid o' \notin leaves(subtree(o, R))\}\ . \end{aligned}$$

The following theorem allows a transition from the clausal structure to the conjunction of atoms. Its proof follows directly from the definition of RInit and RStep.

Theorem 8. *The set of Horn clauses $\mathcal{HC}$ is satisfiable if and only if the conjunction C is not satisfiable.*

Proof tree

Our algorithm attempts to compute a proof tree P that proves unsatisfiability of C using the proof rules presented in Section 2.2. If no proof can be found then our algorithm reports that no solution exists.

Annotated proof tree

We construct a solution for the given Horn clauses through an iterative process, where the intermediate results are called *partial solutions*. Each partial solution is parameterized by a constraint F. An F-partial solution Π for the resolution tree R is a function from nodes of the resolution tree, $nodes(R)$, to constraints that satisfies the following conditions.

$$(\forall o \in leaves(R) : \models o \to \Pi(o)) \wedge \tag{PS1}$$

$$(\forall (o^1, \ldots, o^m, o) \in R : \models \Pi(o^1) \wedge \cdots \wedge \Pi(o^m) \to \Pi(o)) \wedge \tag{PS2}$$

$$(\models \Pi(false) \to F) \wedge \tag{PS3}$$

$$(\forall o \in nodes(R) :\ Smb(\Pi(o)) \subseteq (InSmb(o) \cap OutSmb(o)) \cup Smb(F)) \tag{PS4}$$

Given the proof tree P, we annotate its nodes with partial solutions using the rules and auxiliary functions shown in Figure 8.3. Our annotation uses constraints of the form of solution constraints, introduced in Section 6.3. The rule HcHyp annotates each leaf of the proof tree with the result of applying the function SolHyp. The annotation is enclosed by a pair of square brackets. The rule HcComb shows how to annotate a parent node when provided with an annotation of its children in case when the parent was obtained by a non-negatively weighted sum. The parent annotation is computed by SolComb. Similarly to HcComb, the rule HcCong annotates parent nodes obtained by the congruence rule.

We annotate P and obtain an annotated proof tree A. Our algorithm HcSolve uses the annotation of the root of A to derive a solution for the Horn clauses $\mathcal{HC}$.

$$\textsc{HcHyp}\ \frac{}{t \le 0\ [\ \textsc{SolHyp}(t \le 0)\ ]} \qquad \textsc{HcComb}\ \frac{t_1 \le 0\ [\ \Pi_1\ ] \quad \dots \quad t_n \le 0\ [\ \Pi_n\ ]}{\lambda_1 t_1 + \cdots + \lambda_n t_n \le 0\ [\ \textsc{SolComb}(\Pi_1, \dots, \Pi_n, \lambda_1, \dots, \lambda_n)\ ]}$$

$$\textsc{HcCong}\ \frac{\begin{array}{cc} t_1 - s_1 \le 0\ [\ \Pi_1\ ] & s_1 - t_1 \le 0\ [\ \Pi'_1\ ] \\ \vdots & \vdots \\ t_n - s_n \le 0\ [\ \Pi_n\ ] & s_n - t_n \le 0\ [\ \Pi'_n\ ] \end{array}}{f(t_1, \dots, t_n) - f(s_1, \dots, s_n) \le 0\ [\textsc{SolCong}(f(t_1, \dots, t_n), f(s_1, \dots, s_n), \Pi_1, \dots, \Pi_n, \Pi'_1, \dots, \Pi'_n)]}$$

function SolHyp
input
  $t \le 0$: inequality term/node in R
begin
  for each $o \in \mathit{nodes}(R)$ **do**
    if $t \le 0 \in \mathit{leaves}(\mathit{subtree}(o, R))$ **then**
      $\Pi(o) := \langle [], t \rangle$
    else
      $\Pi(o) := \langle [], 0 \rangle$
  return Π
end

function SolComb
input
  $\Pi_1, \dots, \Pi_n$: partial solutions
  $\lambda_1, \dots, \lambda_n$: constants
begin
  for each $o \in \mathit{nodes}(R)$ **do**
    for each $i \in 1..n$ **do**
      $\langle L_i, t_i \rangle := \Pi_i(o)$
    $\Pi(o) := \langle L_1 \bullet \cdots \bullet L_n, \lambda_1 t_1 + \cdots + \lambda_n t_n \rangle$
  return Π
end

function SolCong
input
  $f(t_1, \dots, t_n),\ f(s_1, \dots, s_n)$: terms
  $\Pi_1, \dots, \Pi_n,\ \Pi'_1, \dots, \Pi'_n$: partial solutions
begin
  for each $o \in \mathit{nodes}(R)$ **do**
    for each $i \in 1..n$ **do**
      $\langle L_i, p_i \rangle := \Pi_i(o)$
      $\langle L'_i, p'_i \rangle := \Pi'_i(o)$
    $(C, D, p) :=$
        match $\mathit{Smb}(f(t_1, \dots, t_n)) \subseteq \mathit{OutSmb}(o), \mathit{Smb}(f(s_1, \dots, s_n)) \subseteq \mathit{OutSmb}(o)$ **with**
          | $\mathit{true}, \mathit{true}$ -> ($\bigwedge_{i=1}^{n}(p_i \le 0 \wedge p'_i \le 0),\ \mathit{true},\ 0$)
          | $\mathit{true}, \mathit{false}$ -> ($\bigwedge_{i=1}^{n} p_i + p'_i \le 0,\ \bigwedge_{i=1}^{n} -p_i - p'_i \le 0,\ f(s_1 + p_1, \dots, s_n + p_n) - f(s_1, \dots, s_n)$)
          | $\mathit{false}, \mathit{true}$ -> ($\bigwedge_{i=1}^{n} p_i + p'_i \le 0,\ \bigwedge_{i=1}^{n} -p_i - p'_i \le 0,\ f(t_1, \dots, t_n) - f(t_1 + p'_1, \dots, t_n + p'_n)$)
          | $\mathit{false}, \mathit{false}$ -> ($\mathit{true},\ \bigwedge_{i=1}^{n}(t_i - s_i - p_i \le 0 \wedge s_i - t_i - p'_i \le 0),\ f(t_1, \dots, t_n) - f(s_1, \dots, s_n)$)
    $\Pi(o) := \langle L_1 \bullet \cdots \bullet L_n \bullet L'_1 \bullet \cdots \bullet L'_n \bullet (C, D), p \rangle$
  return Π
end

Figure 8.3: Rules for annotating a resolution tree R.

8.3 Correctness and complexity

This section presents the correctness and complexity properties of our algorithm and provides the corresponding proofs.

The correctness of our algorithm follows from Proposition 1 and Theorems 9–11 below. First, we establish that a $1 \leq 0$-partial solution, which satisfies Equations (PS1)–(PS4), defines a solution for the given Horn clauses.

Theorem 9. *$1 \leq 0$-partial solution defines a solution of the Horn clauses.*

Proof. Due to (PS1)–(PS3), a $1 \leq 0$-partial solution satisfies the Horn clauses. Since, $\mathit{Smb}(1 \leq 0)$ is empty, (PS4) is equivalent to the restriction on symbols appearance for a solution of the Horn clauses. ☐

Now, we show that the annotations computed by the rules in Figure 8.3 satisfy the partial solution conditions in Equations (PS1)–(PS4). This step relies on the following inductive invariant.

Definition 4 ($t \leq 0$-annotation invariant)**.** *Π is $t \leq 0$-annotation invariant for the resolution tree R if there exists $r \geq 0$ such that for each $o \in \mathit{nodes}(R)$ the following conditions hold.*

- *$\Pi(o)$ is a solution constraint such that*

$$\Pi(o) = \langle((C_1, D_1), \dots, (C_r, D_r)), p\rangle. \tag{AI-1}$$

- *If $o \in \mathit{leaves}(R)$ then*

$$\left(\forall i \in 1..r : \quad \models o \wedge \bigwedge_{k=1}^{i-1} D_k \rightarrow C_i\right) \wedge \tag{AI-2a}$$

$$\left(\models o \wedge \bigwedge_{k=1}^{r} D_k \rightarrow p \leq 0\right). \tag{AI-2b}$$

- *If $(o^1, \dots, o^m, o) \in R$ and $\forall j \in 1..m : \Pi(o^j) = \langle((C^j_1, D^j_1), \dots, (C^j_r, D^j_r)), p^j\rangle$ then*

$$\left(\forall i \in 1..r : \quad \models \left(\bigwedge_{k=1}^{i} \bigwedge_{l=1}^{m} C^l_k\right) \ \wedge \ \bigwedge_{k=1}^{i-1} D_k \rightarrow C_i\right) \wedge \tag{AI-3a}$$

$$\left(\begin{matrix}\forall i \in 1..r \\ \forall j \in 1..m\end{matrix} : \quad \models \begin{matrix}\left(\bigwedge_{l \in 1..m \setminus \{j\}} C^l_i\right) \wedge \\ \left(\bigwedge_{k=1}^{i-1} \bigwedge_{l=1}^{m} C^l_k\right) \wedge \bigwedge_{k=1}^{i} D_k \rightarrow D^j_i\end{matrix}\right) \wedge \tag{AI-3b}$$

$$\left(\models \left(\bigwedge_{k=1}^{r} \bigwedge_{l=1}^{m} C^l_k\right) \ \wedge \ \bigwedge_{k=1}^{r} D_k \rightarrow p - p^1 - \dots - p^m \leq 0\right). \tag{AI-3c}$$

- *If $o = \mathit{false}$ then*

$$p = t \wedge \forall i \in 1..r : D_i = C_i = \mathit{true}. \tag{AI-4}$$

- *Conditions on symbol appearance:*

$$\mathit{Smb}(\{C_1, \dots, C_r, D_1, \dots, D_r, p \leq 0\}) \subseteq \mathit{InSmb}(o) \wedge \tag{AI-5}$$

$$\mathit{Smb}(\{C_1, \dots, C_r, D_1, \dots, D_r, t - p \leq 0\}) \subseteq \mathit{OutSmb}(o). \tag{AI-6}$$

Theorem 10. *Each $t \leq 0$-annotation invariant is a $t \leq 0$-partial solution.*

Proof. Let Π be a $t \leq 0$-annotation invariant and let $o \in \mathit{nodes}(R)$. Then, $\Pi(o)$ satisfies (AI-1)–(AI-6). We will prove that Π is $t \leq 0$-partial solution by showing (PS3),(PS4), (PS1), and (PS2).

(PS3)**:** If $o = \mathit{false}$ then (AI-4) directly implies (PS3).

(PS4)**:** Due to (AI-5), $\mathit{Smb}(\Pi(o)) \subseteq \mathit{InSmb}(o)$. Due to (AI-6), $\mathit{Smb}(t - p \leq 0) \subseteq \mathit{OutSmb}(o)$. Now, let us assume there is a subterm s in p such that $\mathit{Smb}(s) \nsubseteq \mathit{OutSmb}(o) \cup \mathit{Smb}(t \leq 0)$ and s does not have $+$ as the outermost function symbol. Therefore, s must be a subterm of $t-p$. Therefore, $\mathit{Smb}(t-p \leq 0) \nsubseteq \mathit{OutSmb}(o)$. Hence, we obtain a contradiction. Therefore, $\mathit{Smb}(p \leq 0) \subseteq \mathit{OutSmb}(o) \cup \mathit{Smb}(t \leq 0)$. So we deduce $\mathit{Smb}(\Pi(o)) \subseteq \mathit{InSmb}(o) \cap (\mathit{OutSmb}(o) \cup \mathit{Smb}(t \leq 0))$. Hence, (PS4) holds.

(PS1)**:** Let $o \in \mathit{leaves}(R)$. First, we will prove the following validity for all $i \in 0..r$ by induction.

$$\models o \wedge \textstyle\bigwedge_{k=1}^{r-i} D_k \rightarrow \langle(\ (C_{r-i+1}, D_{r-i+1}), \ldots, (C_r, D_r)\ ), p\rangle$$

Base case: $i = 0$. (AI-2b) implies $\models o \wedge \bigwedge_{k=1}^{r} D_k \rightarrow \langle(), p\rangle$.
Induction step: $r > i > 0$. By induction hypothesis, we have

$$\models o \wedge \textstyle\bigwedge_{k=1}^{r-i} D_k \rightarrow \langle(\ (C_{r-i+1}, D_{r-i+1}), \ldots, (C_r, D_r)\ ), p\rangle.$$

By separating D_{r-i}, we obtain

$$\models o \wedge \textstyle\bigwedge_{k=1}^{r-i-1} D_k \rightarrow (D_{r-i} \rightarrow \langle(\ (C_{r-i+1}, D_{r-i+1}), \ldots, (C_r, D_r)\ ), p\rangle).$$

Due to the (AI-2a), $\models o \wedge \bigwedge_{k=1}^{r-i-1} D_k \rightarrow C_{r-i}$. Therefore,

$$\models o \wedge \textstyle\bigwedge_{k=1}^{r-i} D_k \rightarrow (C_{r-i} \wedge (D_{r-i} \rightarrow \langle(\ (C_{r-i+1}, D_{r-i+1}), \ldots, (C_r, D_r)\ ), p\rangle)),$$

which is equivalent to

$$\models o \wedge \textstyle\bigwedge_{k=1}^{r-i-1} D_k \rightarrow \langle(\ (C_{r-i}, D_{r-i}), \ldots, (C_r, D_r)\ ), p\rangle.$$

From our proved validity, we obtain for $i = r$:

$$\models o \rightarrow \langle(\ (C_1, D_1), \ldots, (C_r, D_r)\ ), p\rangle.$$

Hence, (PS1) holds.

(PS2)**:** Let $(o^1, \ldots, o^m, o) \in R$. First, we will prove the following validity for all $i \in 0..r$ by induction.

$$\begin{aligned} \models \quad & \textstyle\bigwedge_{j=1}^{m} \langle(\ (C^j_{r-i+1}, D^j_{r-i+1}), \ldots, (C^j_r, D^j_r)\ ), p^j\rangle \wedge \\ & \left(\textstyle\bigwedge_{k=1}^{r-i} \bigwedge_{l=1}^{m} C^l_k\right) \wedge\ \textstyle\bigwedge_{k=1}^{r-i} D_k \rightarrow \langle(\ (C_{r-i+1}, D_{r-i+1}), \ldots, (C_r, D_r)\ ), p\rangle \end{aligned}$$

Base case: $i = 0$. (AI-3c) implies

$$\models \textstyle\bigwedge_{j=1}^{m} \langle(), p^j\rangle\ \wedge\ \left(\bigwedge_{k=1}^{r} \bigwedge_{l=1}^{m} C^l_k\right) \wedge\ \bigwedge_{k=1}^{r} D_k \rightarrow \langle(), p\rangle,$$

which is the base case.
Induction step: $r > i > 0$. Consider the left hand side of induction step $i + 1$,

$$\textstyle\bigwedge_{j=1}^{m} \langle(\ (C^j_{r-i}, D^j_{r-i}), \ldots, (C^j_r, D^j_r)\ ), p^j\rangle\ \wedge\ \left(\bigwedge_{k=1}^{r-i-1} \bigwedge_{l=1}^{m} C^l_k\right) \wedge\ \bigwedge_{k=1}^{r-i-1} D_k.$$

By unfolding definition of a solution constraint once,

$$\begin{aligned} & \textstyle\bigwedge_{j=1}^{m} (D^j_{r-i} \rightarrow \langle(\ (C^j_{r-i+1}, D^j_{r-i+1}), \ldots, (C^j_r, D^j_r)\ ), p^j\rangle) \wedge \\ & \left(\textstyle\bigwedge_{k=1}^{r-i} \bigwedge_{l=1}^{m} C^l_k\right) \wedge\ \textstyle\bigwedge_{k=1}^{r-i-1} D_k. \end{aligned}$$

Due to (AI-3a), the above formula implies C_{r-i}.
Now lets take conjunction of the above formula and D_{r-i},

$$\bigwedge_{j=1}^{m}(D^j_{r-i} \to \langle(\,(C^j_{r-i+1}, D^j_{r-i+1}), \ldots, (C^j_r, D^j_r)\,), p^j\rangle) \wedge \\ \left(\bigwedge_{k=1}^{r-i} \bigwedge_{l=1}^{m} C^l_k\right) \wedge\ \bigwedge_{k=1}^{r-i} D_k.$$

Due to (AI-3b), the above formula implies

$$\bigwedge_{j=1}^{m}(D^j_{r-i} \to \langle(\,(C^j_{r-i+1}, D^j_{r-i+1}), \ldots, (C^j_r, D^j_r)\,), p^j\rangle) \wedge \\ \left(\bigwedge_{k=1}^{r-i} \bigwedge_{l=1}^{m} C^l_k\right) \wedge\ \bigwedge_{k=1}^{r-i} D_k \wedge \bigwedge_{j=1}^{m} D^j_{r-i}.$$

Therefore,

$$\bigwedge_{j=1}^{m}\langle(\,(C^j_{r-i+1}, D^j_{r-i+1}), \ldots, (C^j_r, D^j_r)\,), p^j\rangle \wedge \\ \left(\bigwedge_{k=1}^{r-i} \bigwedge_{l=1}^{m} C^l_k\right) \wedge\ \bigwedge_{k=1}^{r-i} D_k.$$

Due to the induction hypothesis, the above formula implies

$$\langle(\,(C_{r-i+1}, D_{r-i+1}), \ldots, (C_r, D_r)\,), p\rangle.$$

So, we have proven that the left hand side of the induction step at $i+1$ implies

$$C_{r-i} \wedge (D_{r-i} \to \langle(\,(C_{r-i+1}, D_{r-i+1}), \ldots, (C_r, D_r)\,), p\rangle),$$

which is the right hand side of the induction step at $i+1$.

From our proved validity, we obtain for $i = r$,

$$\models \bigwedge_{j=1}^{m}\langle(\,(C^j_1, D^j_1), \ldots, (C^j_r, D^j_r)\,), p^j\rangle \to \langle(\,(C_1, D_1), \ldots, (C_r, D_r)\,), p\rangle.$$

Hence, (PS2) holds. □

The following three lemmas will be used to prove Theorem 11.

Lemma 4. *Let* Π *be* $t \leq 0$*-annotation invariant and let* Π' *be* $t' \leq 0$*-annotation invariant. Let* Π_1 *and* Π_1 *be a function from* R *to constraints such that*

$$\forall o \in \mathit{nodes}(R) : \Pi(o) = \langle L, p\rangle \wedge \Pi'(o) = \langle L', \_\rangle \to \Pi_1(o) = \langle L \bullet L', p\rangle$$

and

$$\forall o \in \mathit{nodes}(R) : \Pi(o) = \langle L, p\rangle \wedge \Pi'(o) = \langle L', \_\rangle \to \Pi_2(o) = \langle L' \bullet L, p\rangle.$$

Π_1 *and* Π_2 *are* $t \leq 0$*-annotation invariants.*

Proof. We will only deal with Π_1. The proof for Π_2 is similar.

Let $o \in \mathit{nodes}(R)$, $\Pi(o) = \langle(\,(C_1, D_1), \ldots, (C_n, D_n)\,), p\rangle$, and $\Pi'(o) = \langle(\,(C_{n+1}, D_{n+1}), \ldots, (C_{n+m}, D_{n+m})\,), \_\rangle$. Then, $\Pi_1(o) = \langle(\,(C_1, D_1), \ldots, (C_{n+m}, D_{n+m})\,), p\rangle$. $\Pi_1(o)$ maps to a solution constraint that has prefix sequence of length $n+m$. Therefore, (AI-1) holds. (AI-2a)–(AI-3c) for $\Pi_1(o)$ are satisfied since these conditions have stronger left hand sides compare to the corresponding conditions for $\Pi(o)$ and $\Pi'(o)$. (AI-4)–(AI-6) are directly holds. □

The above lemma can be applied multiple times on a $t \leq 0$-annotation invariant satisfying Π to show that a prefix extension in the above way does not violate $t \leq 0$-annotation invariant.

Lemma 5. *Let* $(o^1, \ldots, o^m, o) \in R$. *If* $\mathit{Smb}(f(t_1, \ldots, t_n)) \subseteq \mathit{OutSmb}(o)$ *then* $\forall l \in 1..m : \mathit{Smb}(f(t_1, \ldots, t_n)) \subseteq \mathit{OutSmb}(o^l)$.

The proof of above lemma is left for the reader to verify.

Lemma 6. *Let* $(o^1, \ldots, o^m, o) \in R$. *If* $Smb(f(t_1, \ldots, t_n)) \nsubseteq OutSmb(o)$ *then either of the following cases is true.*

(1) $\forall l \in 1..m : \ Smb(f(t_1, \ldots, t_n)) \subseteq OutSmb(o^l)$

(2) $\exists j : \quad Smb(f(t_1, \ldots, t_n)) \nsubseteq OutSmb(o^j) \wedge$
$\forall l \in 1..m \setminus \{j\} : \ Smb(f(t_1, \ldots, t_n)) \subseteq OutSmb(o^l)$.

Proof. Since PComb does not allow introduction of terms that are not present in the input atoms, if $Smb(f(t_1, \ldots, t_n)) \nsubseteq OutSmb(o)$ then $Smb(f(t_1, \ldots, t_n)) \subseteq InSmb(o)$ and there exist at least one child node o^j such that $Smb(f(t_1, \ldots, t_n)) \subseteq InSmb(o^j)$.

If there are at least two children o^{j_1} and o^{j_2} such that $Smb(f(t_1, \ldots, t_n)) \subseteq InSmb(o^{j_1})$ and $Smb(f(t_1, \ldots, t_n)) \subseteq InSmb(o^{j_2})$ then first case will be true.

If there is exactly one child o^j such that $Smb(f(t_1, \ldots, t_n)) \subseteq InSmb(o^{j_1})$ then second case will be true. ☐

Theorem 11. *The annotation rules in Figure 8.3 compute annotation invariants.*

Proof. We will proof that HcHyp computes annotation invariants as base case and HcComb and HcCong inductively compute the annotation invariants.

HcHyp rule: Let $\Pi = \textsc{SolHyp}(t \leq 0)$. For each $o \in R$, $\Pi(o)$ is $\langle [], p \rangle$, which implies $r = 0$ in the Definition 4 with respect to Π. Therefore, (AI-1), (AI-2a), (AI-3a), and (AI-3b) hold, trivially.

Let $o \in leaves(R)$. (AI-2b) holds since if $o = (t \leq 0)$ then $p = t$ else $p = 0$.

Let $(o^1, \ldots, o^m, o) \in R$. If $(t \leq 0)$ is in the subtree of the node o then $p = t$. Since R is a tree, there is $j \in 1..m$ such that the subtree of o^j contains $(t \leq 0)$. Therefore, $p^j = t$ and $\forall l \in 1..m \setminus \{j\} : p^l = 0$. Therefore, the right hand side of (AI-3b) is $0 \leq 0$. In other case,i.e., $t \leq 0$ is not in subtree of node o, $p = 0$ and $\forall j \in 1..m : p^l = 0$. Again the right hand side of (AI-3b) is $0 \leq 0$. Therefore, in both the cases (AI-3b) holds.

Since all leaves are in the subtree rooted at the node *false*, (AI-4) is satisfied.

If $(t \leq 0)$ is in the subtree of o then $p = t$. Hence, $p - t = 0$. Therefore (AI-5) and (AI-6) hold. Otherwise, i.e., if $(t \leq 0)$ is not in the subtree of o, then $p = 0$. Hence, $p - t = -t$. Therefore (AI-5) and (AI-6) holds.

HcComb rule: By the induction hypothesis, Π_i is $t_i \leq 0$-annotation invariant for each $i \in 1..n$. Let $\Pi = \textsc{SolComb}(\Pi_1, \ldots, \Pi_n, \lambda_1, \ldots, \lambda_n)$. We show that Π is $\lambda_1 t_1 + \cdots + \lambda_n t_n \leq 0$-annotation invariant. For each $i \in 1..n$, we first construct $\overline{\Pi}_i$ such that

$$\forall o \in nodes(R) : \begin{pmatrix} \Pi_1(o) = \langle L_1, p_1 \rangle \\ \wedge \\ \vdots \\ \wedge \\ \Pi_n(o) = \langle L_n, p_n \rangle \end{pmatrix} \rightarrow \overline{\Pi}_i(o) = \langle L_1 \bullet \cdots \bullet L_n, p_i \rangle.$$

Due to Lemma 4, $\overline{\Pi}_i$ is $t_i \leq 0$-annotation invariant. SolComb constructs Π such that

$$\forall o \in nodes(R) : \begin{pmatrix} \overline{\Pi}_1(o) = \langle L, p_1 \rangle \\ \wedge \\ \vdots \\ \wedge \\ \overline{\Pi}_n(o) = \langle L, p_n \rangle \end{pmatrix} \rightarrow \Pi(o) = \langle L, \lambda_1 p_1 + \cdots + \lambda_n p_n \rangle.$$

(AI-1), (AI-2a), (AI-3a), (AI-3b), and (AI-4) w.r.t. $\lambda_1 t_1 + \cdots + \lambda_n t_n \leq 0$-annotation invariant are trivially satisfied.

Let $o \in \mathit{leaves}(R)$. The left hand sides of (AI-2b) w.r.t. $\overline{\Pi}_1(o), \ldots, \overline{\Pi}_n(o)$ are equal and they also equal to the left hand side of (AI-2b) w.r.t. $\Pi(o)$. The right hand side of (AI-2b) w.r.t. $\Pi(o)$ is a linear combination of the right hand sides of (AI-2b) w.r.t. $\overline{\Pi}_1(o), \ldots, \overline{\Pi}_n(o)$. Therefore, (AI-2b) w.r.t. $\Pi(o)$ holds. A similar argument proves (AI-3c). $\mathit{Smb}(\{p_1 \leq 0, \ldots, p_n \leq 0\}) \subseteq \mathit{InSmb}(o)$, therefore $\mathit{Smb}(\lambda_1 p_1 + \cdots + \lambda_n p_n) \subseteq \mathit{InSmb}(o)$. Hence, (AI-5) holds. A similar argument proves (AI-6).

HcCong rule: By the induction hypothesis, Π_i is $t_i - s_i \leq 0$-annotation invariant and Π'_i is $t_i - s_i \leq 0$-annotation invariant for $i \in 1..n$. Let $\Pi = \textsc{SolCong}(f(t_1, \ldots, t_n),\ f(s_1, \ldots, s_n), \Pi_1, \ldots, \Pi_n, \Pi'_1, \ldots, \Pi'_n)$. We prove that Π is $f(t_1, \ldots, t_n) - f(s_1, \ldots, s_n) \leq 0$-annotation invariant. For each $i \in 1..n$, we construct $\overline{\Pi}_i$ and $\overline{\Pi'}_n$ such that

$$\forall o \in \mathit{nodes}(R) : \begin{pmatrix} \Pi_1(o) = \langle L_1, p_1 \rangle \\ \wedge \\ \vdots \\ \wedge \\ \Pi_n(o) = \langle L_n, p_n \rangle \\ \wedge \\ \Pi'_1(o) = \langle L'_1, p'_1 \rangle \\ \wedge \\ \vdots \\ \wedge \\ \Pi'_n(o) = \langle L'_n, p'_n \rangle \end{pmatrix} \rightarrow \begin{pmatrix} \overline{\Pi}_i(o) = \langle & L_1 \bullet \cdots \bullet L_n \bullet \\ & L'_1 \bullet \cdots \bullet L'_n,\ p_i \rangle \\ & \wedge \\ \overline{\Pi'}_i(o) = \langle & L_1 \bullet \cdots \bullet L_n \bullet \\ & L'_1 \bullet \cdots \bullet L'_n,\ p'_i \rangle \end{pmatrix}.$$

Due to Lemma 4, $\overline{\Pi}_i$ satisfies $t_i - s_i \leq 0$-annotation invariant and $\overline{\Pi'}_i$ satisfies $s_i - t_i \leq 0$-annotation invariant for $i \in 1..n$.

Let $o \in \mathit{nodes}(R)$. Let $\overline{\Pi}_i(o) = \langle ((C_1, D_1), \ldots, (C_r, D_r)), p_i \rangle$ and let $\overline{\Pi'_i}(o) = \langle ((C_1, D_1), \ldots, (C_r, D_r)), p'_i \rangle$ for each $i \in 1..n$. $\textsc{SolCong}$ returns Π such that $\Pi(o) = \langle ((C_1, D_1), \ldots, (C_r, D_r), (C_{r+1}, D_{r+1})), p \rangle$, where C_{r+1}, D_{r+1} and p are computed at line 5. At line 6 of function $\textsc{SolCong}$, match has four cases which we will lead to four or more cases distinction for proving (AI-1)–(AI-6) w.r.t. $f(t_1, \ldots, t_n) - f(s_1, \ldots, s_n) \leq 0$-annotation invariant. Now rest of the proof is divided into proving each of the conditions.

(AI-1): Since Π maps all nodes of R to solution constraints that have prefix sequence of length $r + 1$, (AI-1) holds.

(AI-5) and (AI-6): We show in the following four cases that C_{r+1}, D_{r+1}, and p satisfy (AI-5) and (AI-6).

(1) $\mathit{Smb}(f(t_1, \ldots, t_n)) \subseteq \mathit{OutSmb}(o) \wedge \mathit{Smb}(f(s_1, \ldots, s_n)) \subseteq \mathit{OutSmb}(o)$:
Let $i \in 1..n$. Due to the condition of this case, $\mathit{Smb}(t_i - s_i) \subseteq \mathit{OutSmb}(o)$. (AI-6) w.r.t. $\overline{\Pi_i}(o)$ implies $\mathit{Smb}(t_i - s_i - p_i) \subseteq \mathit{OutSmb}(o)$. Therefore, $\mathit{Smb}(p_i) \subseteq \mathit{OutSmb}(o)$. Due to (AI-5) w.r.t. $\overline{\Pi_i}(o)$, $\mathit{Smb}(p_i) \subseteq \mathit{InSmb}(o)$. A similar argument proves $\mathit{Smb}(p'_i) \subseteq \mathit{OutSmb}(o)$ and $\mathit{Smb}(p'_i) \subseteq \mathit{InSmb}(o)$. Therefore, C_{r+1} satisfies (AI-5) and (AI-6) w.r.t. $\Pi(o)$. Since, $D_{r+1} = \mathit{true}$ and $p = 0$, we do not need to prove anything for them.

(2) $\mathit{Smb}(f(t_1, \ldots, t_n)) \subseteq \mathit{OutSmb}(o) \wedge \mathit{Smb}(f(s_1, \ldots, s_n)) \not\subseteq \mathit{OutSmb}(o)$:
Let $i \in 1..n$. Due to (AI-5) w.r.t. $\overline{\Pi_i}(o)$ and $\overline{\Pi'_i}(o)$, $\mathit{Smb}(p_i + p'_i) \in \mathit{InSmb}(o)$. Due to (AI-6), $\mathit{Smb}(t_i - s_i - p_i) \in \mathit{OutSmb}(o)$ and $\mathit{Smb}(s_i - t_i - p'_i) \in \mathit{OutSmb}(o)$ therefore $\mathit{Smb}(-p_i - p'_i) \in \mathit{OutSmb}(o)$. Therefore, C_{r+1} and D_{r+1} satisfy (AI-5) and (AI-6) of Π.

$\mathit{Smb}(f(s_1, \ldots, s_n)) \not\subseteq \mathit{OutSmb}(o)$ implies $\mathit{Smb}(f(s_1, \ldots, s_n)) \subseteq \mathit{InSmb}(o)$. Therefore, $\mathit{Smb}(s_i) \subseteq \mathit{InSmb}(o)$. Therefore, $\mathit{Smb}(s_i + p_i) \subseteq \mathit{InSmb}(o)$. Therefore, $\mathit{Smb}(f(s_1 + p_1, \ldots, s_n + p_n) - f(s_1, \ldots, s_n)) \subseteq \mathit{InSmb}(o)$. Hence, (AI-5) w.r.t. $\Pi(o)$ holds. Due to conditions (AI-6) w.r.t. $\overline{\Pi_i}(o)$, $\mathit{Smb}(t_i - s_i - p_i) \subseteq \mathit{OutSmb}(o)$. Since $\mathit{Smb}(t_i) \subseteq \mathit{OutSmb}(o)$, $\mathit{Smb}(s_i + p_i) \subseteq \mathit{OutSmb}(o)$. Therefore,

$Smb(f(t_1,\ldots,t_n) - f(s_1+p_1,\ldots,s_n+p_n)) \subseteq OutSmb(o)$. Hence, (AI-6) w.r.t. $\Pi(o)$ holds.

(3) $Smb(f(t_1,\ldots,t_n)) \nsubseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \subseteq OutSmb(o)$:
A similar argument as in the previous case.

(4) $Smb(f(t_1,\ldots,t_n)) \nsubseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o)$:
Due to the condition of this case, $Smb(f(t_1,\ldots,t_n) - f(s_1,\ldots,s_n)) \subseteq InSmb(o)$. Hence, p satisfies (AI-5) and (AI-6) w.r.t. $\Pi(o)$. Let $i \in 1..n$. Due to (AI-6) w.r.t. $\overline{\Pi_i}(o)$ and $\overline{\Pi_i'}(o)$, $Smb(t_i - s_i - p_i, s_i - t_i - p_i') \subseteq OutSmb(o)$. Due to (AI-5) w.r.t. $\overline{\Pi_i}(o)$ and $\overline{\Pi_i'}(o)$, $Smb(p_i, p_i') \subseteq InSmb(o)$. Due to the condition of this case, $Smb(t_i - s_i) \subseteq InSmb(o)$. Therefore, $Smb(t_i - s_i - p_i, s_i - t_i - p_i') \subseteq InSmb(o)$. Hence, D_{r+1} satisfies (AI-5) and (AI-6) w.r.t. $\Pi(o)$. Since $C_{r+1} = true$, we do not have to prove anything for it.

(AI-2a) and (AI-2b): Let $o \in leaves(R)$. In (AI-2a) w.r.t. $\Pi(o)$, the implications for $i \in 1..r$ are satisfied due to (AI-2a) w.r.t. $\overline{\Pi_1}(o)$ and we only prove $r+1^{\text{th}}$ instantiation of the implications, i.e.,

$$\models o \wedge \ \bigwedge_{k=1}^{r} D_k \rightarrow C_{r+1}. \tag{8.6}$$

We also prove condition (AI-2b) w.r.t. $\Pi(o)$. There are again four cases.

(1) $Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \subseteq OutSmb(o)$
Since $C_{r+1} = \bigwedge_{i=1}^{n}(p_i \leq 0 \wedge p_i' \leq 0)$, (AI-2b) w.r.t. $\overline{\Pi_i}(o)$ and $\overline{\Pi_i'}(o)$ imply (8.6). (AI-2b) w.r.t. $\Pi(o)$ is trivially satisfied.

(2) $Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o)$
Since $C_{r+1} = \bigwedge_{i=1}^{n}(p_i + p_i' \leq 0)$, (AI-2b) w.r.t. $\overline{\Pi_i}(o)$ and $\overline{\Pi_i'}(o)$ imply (8.6). In this case, $D_{r+1} = \bigwedge_{i=1}^{n}(-p_i - p_i' \leq 0)$. Let $i \in 1..n$. The left hand side of (AI-2b) w.r.t. $\Pi(o)$ implies $-p_i - p_i' \leq 0 \wedge p_i' \leq 0 \wedge p_i \leq 0$. So, $p_i = 0$. Therefore, $s_i + p_i = s_i$. Therefore, $f(s_1+p_1,\ldots,s_n+p_n) - f(s_1,\ldots,s_n) \leq 0$, which is the right hand side of (AI-2b) w.r.t. $\Pi(o)$. Hence, (AI-2b) w.r.t. $\Pi(o)$ holds.

(3) $Smb(f(t_1,\ldots,t_n)) \nsubseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \subseteq OutSmb(o)$
A similar argument as in the previous case.

(4) $Smb(f(t_1,\ldots,t_n)) \nsubseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o)$
In this case, $C_{r+1} = true$ and $D_{r+1} = \bigwedge_{i=1}^{n}(t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p_i' \leq 0)$. (8.6) is trivially satisfied. Left hand sides of (AI-2b) w.r.t. $\overline{\Pi_i}$ and $\overline{\Pi_i'}$ are equal, and their conjunction with D_{r+1} is equal to the left hand side of (AI-2b) w.r.t. $\Pi(o)$. Therefore, the left hand side of (AI-2b) w.r.t. $\Pi(o)$ implies $\bigwedge_{i=1}^{n}(t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p_i' \leq 0) \wedge \bigwedge_{i=1}^{n}(p_i \leq 0 \wedge p_i' \leq 0)$. Therefore, $\bigwedge_{i=1}^{n}(t_i - s_i \leq 0 \wedge s_i - t_i \leq 0)$. Therefore, $\bigwedge_{i=1}^{n} t_i = s_i$. Therefore, $f(t_1,\ldots,t_n) - f(s_1,\ldots,s_n) \leq 0$, which is the right hand side of (AI-2b) w.r.t. $\Pi(o)$. Hence, (AI-2b) w.r.t. $\Pi(o)$ holds.

(AI-3a), (AI-3b) and (AI-3c): Let $(o^1,\ldots,o^m,o) \in R$. For each $l \in 1..m$, let $\overline{\Pi_i}(o^l) = \langle(\ (C_1^l, D_1^l),\ldots,(C_r^l, D_r^l)\ ), p_i^l\rangle$, $\overline{\Pi_i'}(o^l) = \langle(\ (C_1^l, D_1^l),\ldots,(C_r^l, D_r^l)\ ), p_i^{l\prime}\rangle$, and $\overline{\Pi}(o^l) = \langle(\ (C_1^l, D_1^l),\ldots,(C_r^l, D_r^l)\ ), p^l\rangle$. In (AI-3a) w.r.t. $\Pi(o)$, the implications for $i \in 1..r$ are satisfied due to (AI-3a) w.r.t. $\overline{\Pi_1}(o)$. We only prove $r+1^{\text{th}}$ instantiation of the implications, i.e.,

$$\left(\bigwedge_{k=1}^{r+1} \bigwedge_{l=1}^{m} C_k^l\right) \ \wedge \ \bigwedge_{k=1}^{r} D_k \rightarrow C_{r+1}$$

By reorganizing the above formula,

$$\bigwedge_{l=1}^{m} C_{r+1}^{l} \wedge \big(\big(\bigwedge_{k=1}^{r} \bigwedge_{l=1}^{m} C_{k}^{l}\big) \quad \wedge \quad \bigwedge_{k=1}^{r} D_{k}\big) \rightarrow C_{r+1}$$

Due to (AI-3b) w.r.t. $\overline{\Pi_1}(o), \ldots, \overline{\Pi_n}(o)$ and $\overline{\Pi'_1}(o), \ldots, \overline{\Pi'_n}(o)$, we need to prove the following formula in order to prove the formula above.

$$\bigwedge_{l=1}^{m} C_{r+1}^{l} \wedge \bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i - p_i^1 - \cdots - p_i^m \leq 0 \wedge \\ p'_i - p_i^{1\prime} - \cdots - p_i^{m\prime} \leq 0 \end{array} \right) \rightarrow C_{r+1} \tag{8.7}$$

In (AI-3b) w.r.t. $\Pi(o)$, the implications for $i \in 1..r$ are satisfied due to (AI-3b) w.r.t. $\overline{\Pi_1}(o)$. We only prove $r+1^{\text{th}}$ instantiations of the implications, i.e.,

$$\begin{array}{ll} \forall j \in 1..m : \models & \Big(\bigwedge_{l \in 1..m \setminus \{j\}} C_{r+1}^{l}\Big) \wedge \\ & \big(\bigwedge_{k=1}^{r} \bigwedge_{l=1}^{m} C_{k}^{l}\big) \wedge \bigwedge_{k=1}^{r+1} D_k \rightarrow D_{r+1}^{j}. \end{array}$$

By reorganizing the above formula,

$$\begin{array}{ll} \forall j \in 1..m : \models & \Big(\bigwedge_{l \in 1..m \setminus \{j\}} C_{r+1}^{l}\Big) \wedge D_{r+1} \wedge \\ & \big(\big(\bigwedge_{k=1}^{r} \bigwedge_{l=1}^{m} C_{k}^{l}\big) \wedge \bigwedge_{k=1}^{r} D_k\big) \rightarrow D_{r+1}^{j}. \end{array}$$

Due to (AI-3b) w.r.t. $\overline{\Pi_1}(o), \ldots, \overline{\Pi_n}(o)$ and $\overline{\Pi'_1}(o), \ldots, \overline{\Pi'_n}(o)$, we need to prove the following formula in order to prove the formula above.

$$\begin{array}{ll} \forall j \in 1..m : \models & \Big(\bigwedge_{l \in 1..m \setminus \{j\}} C_{r+1}^{l}\Big) \wedge D_{r+1} \wedge \\ & \bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i - p_i^1 - \cdots - p_i^m \leq 0 \wedge \\ p'_i - p_i^{1\prime} - \cdots - p_i^{m\prime} \leq 0 \end{array} \right) \rightarrow D_{r+1}^{j} \end{array} \tag{8.8}$$

(AI-3c) w.r.t. $\Pi(o)$ is

$$\models \Big(\bigwedge_{k=1}^{r+1} \bigwedge_{l=1}^{m} C_{k}^{l}\Big) \quad \wedge \quad \bigwedge_{k=1}^{r+1} D_k \rightarrow p - p^1 - \cdots - p^m \leq 0$$

By reorganizing the above formula,

$$\models \bigwedge_{l=1}^{m} C_{r+1}^{l} \wedge D_{r+1} \wedge \big(\big(\bigwedge_{k=1}^{r} \bigwedge_{l=1}^{m} C_{k}^{l}\big) \quad \wedge \quad \bigwedge_{k=1}^{r} D_k\big) \rightarrow p - p^1 - \cdots - p^m \leq 0$$

Due to (AI-3b) w.r.t. $\overline{\Pi_1}(o), \ldots, \overline{\Pi_n}(o)$ and $\overline{\Pi'_1}(o), \ldots, \overline{\Pi'_n}(o)$, we need to prove the following formula in order to prove the formula above.

$$\begin{array}{ll} \models & \bigwedge_{l=1}^{m} C_{r+1}^{l} \wedge D_{r+1} \wedge \\ & \bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i - p_i^1 - \cdots - p_i^m \leq 0 \wedge \\ p'_i - p_i^{1\prime} - \cdots - p_i^{m\prime} \leq 0 \end{array} \right) \rightarrow p - p^1 - \cdots - p^m \leq 0 \end{array} \tag{8.9}$$

We prove (8.7), (8.8), and (8.9) for the following ten cases, which are consequence of Lemmas 5 and 6. In each case, we will present the table of values of C_{r+1}, D_{r+1}, p, and, for each $l \in 1..m$, C_{r+1}^{l}, D_{r+1}^{l} and p^l. Then, provide proves of (8.7), (8.8), and (8.9) for the given values.

(1) $\mathit{Smb}(f(t_1, \ldots, t_n)) \subseteq \mathit{OutSmb}(o) \wedge \mathit{Smb}(f(s_1, \ldots, s_n)) \subseteq \mathit{OutSmb}(o):$

	$\forall l \in 1..m$
$C_{r+1} = \bigwedge_{i=1}^{n}(p_i \leq 0 \wedge p'_i \leq 0)$	$C_{r+1}^{l} = \bigwedge_{i=1}^{n}(p_i^l \leq 0 \wedge p_i^{l\prime} \leq 0)$
$D_{r+1} = \mathit{true}$	$D_{r+1}^{l} = \mathit{true}$
$p = 0$	$p^l = 0$

(8.8) and (8.9) are trivially satisfied. Placing values of C_{r+1}^{l} in left hand side of (8.7), we obtain

$$\bigwedge_{l=1}^{m} \bigwedge_{i=1}^{n}(p_i^l \leq 0 \wedge p_i^{l\prime} \leq 0) \wedge \bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i - p_i^1 - \cdots - p_i^m \leq 0 \wedge \\ p'_i - p_i^{1\prime} - \cdots - p_i^{m\prime} \leq 0 \end{array} \right).$$

By taking linear combination of above atoms, we obtain

$$\bigwedge_{i=1}^{n} \left(\ p_i \le 0 \wedge p_i' \le 0\ \right),$$

which is right hand side of (8.7).

(2) $Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o) \wedge$
$\left(\forall j \in 1..m : \begin{array}{l} Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o^j) \wedge \\ Smb(f(s_1,\ldots,s_n)) \subseteq OutSmb(o^j) \end{array}\right):$

	$\forall l \in 1..m$
$C_{r+1} = \bigwedge_{i=1}^{n}(p_i + p_i' \le 0)$	$C_{r+1}^l = \bigwedge_{i=1}^{n}(p_i^l \le 0 \wedge p_i^{l\prime} \le 0)$
$D_{r+1} = \bigwedge_{i=1}^{n}(-p_i - p_i' \le 0)$	$D_{r+1}^l = true$
$p = f(s_1+p_1,\ldots,s_n+p_n) - f(s_1,\ldots,s_n)$	$p^l = 0$

(8.8) is trivially true. The left hand side of (8.7) is equal to the previous case, therefore, it implies $\bigwedge_{i=1}^{n} \left(\ p_i \le 0 \wedge p_i' \le 0\ \right)$. By taking linear combination of inequalities, we obtain $\bigwedge_{i=1}^{n} (p_i + p_i' \le 0)$, which is the right hand side of (8.7).

In the right hand side of (8.9), $p - p^1 - \cdots - p^n = f(s_1+p_1,\ldots,s_n+p_n) - f(s_1,\ldots,s_n)$. Left hand side of (8.9) implies

$$\bigwedge_{l=1}^{m}\bigwedge_{i=1}^{n}(p_i^l \le 0 \wedge p_i^{l\prime} \le 0) \wedge D_{r+1} \wedge \bigwedge_{i=1}^{n}\left(\begin{array}{l} p_i - p_i^1 - \cdots - p_i^m \le 0 \wedge \\ p_i' - p_i^{1\prime} - \cdots - p_i^{m\prime} \le 0 \end{array}\right).$$

By taking linear combinations, we obtain

$$D_{r+1} \wedge \bigwedge_{i=1}^{n} (p_i \le 0 \wedge p_i' \le 0).$$

After placing value of D_{r+1},

$$\bigwedge_{i=1}^{n}(-p_i - p_i' \le 0) \wedge \bigwedge_{i=1}^{n} (p_i \le 0 \wedge p_i' \le 0).$$

By taking linear combinations, we obtain $\bigwedge_{i=1}^{n}(-p_i \le 0 \wedge p_i \le 0)$. So for all $i \in 1..n$, $p_i = 0$. Therefore, $s_i + p_i = s_i$. Therefore, $f(s_1+p_1,\ldots,s_n+p_n) - f(s_1,\ldots,s_n) \le 0$, which is right hand side of (8.9).

(3) $Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o) \wedge$
$\left(\exists j \in 1..m : \begin{array}{l} Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o^j) \wedge \\ Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o^j) \end{array}\right):$

	$\forall l \in 1..m \setminus \{j\}$
$C_{r+1} = \bigwedge_{i=1}^{n}(p_i + p_i' \le 0)$	$C_{r+1}^l = \bigwedge_{i=1}^{n}(p_i^l \le 0 \wedge p_i^{l\prime} \le 0)$
$D_{r+1} = \bigwedge_{i=1}^{n}(-p_i - p_i' \le 0)$	$D_{r+1}^l = true$
$p = f(s_1+p_1,\ldots,s_n+p_n) - f(s_1,\ldots,s_n)$	$p^l = 0$

$C_{r+1}^j = \bigwedge_{i=1}^{n}(p_i^j + p_i^{j\prime} \le 0)$
$D_{r+1}^j = \bigwedge_{i=1}^{n}(-p_i^j - p_i^{j\prime} \le 0)$
$p^j = f(s_1+p_1^j,\ldots,s_n+p_n^j) - f(s_1,\ldots,s_n)$

Left hand side of (8.7) implies

$$\begin{array}{l} (\bigwedge_{l \in 1..m\setminus\{j\}} \bigwedge_{i=1}^{n}(p_i^l \le 0 \wedge p_i^{l\prime} \le 0)) \wedge \bigwedge_{i=1}^{n}(p_i^j + p_i^{j\prime} \le 0) \wedge \\ \bigwedge_{i=1}^{n}\left(\begin{array}{l} p_i - p_i^1 - \cdots - p_i^m \le 0 \wedge \\ p_i' - p_i^{1\prime} - \cdots - p_i^{m\prime} \le 0 \end{array}\right). \end{array}$$

By taking linear combinations, we obtain $\bigwedge_{i=1}^{n} p_i + p_i' \le 0$, which is right hand side of (8.7).

For (8.8), we only need to prove the instance of implications in which, D^j_{r+1} is equal to $\bigwedge_{i=1}^{n}(-p_i^j - p_i^{j\prime} \leq 0)$. Lets consider left hand side of (8.8), which implies

$$\bigwedge_{i}^{n} \left(\begin{array}{l} \left(\bigwedge_{l\in 1..m\setminus\{j\}} (p_i^l \leq 0 \wedge p_i^{l\prime} \leq 0)\right) \wedge \\ \left(p_i - p_i^1 - \cdots - p_i^m \leq 0 \wedge p_i' - p_i^{1\prime} - \cdots - p_i^{m\prime} \leq 0\right) \wedge -p_i - p_i' \leq 0 \end{array} \right).$$

By by adding above linear inequalities, we can obtain $\bigwedge_{i=1}^{n} -p_i^j - p_i^{j\prime} \leq 0$, which is right hand side of (8.8).

In the right hand side of (8.9), $p - p^1 - \cdots - p^m = f(s_1 + p_1, \ldots, s_n + p_n) - f(s_1 + p_1^j, \ldots, s_n + p_n^j)$. So for proving (8.9), we need to show that the left hand side implies $\bigwedge_{i=0}^{n} s_i + p_i = s_i + p_i^j$. By further simplification, $\bigwedge_{i=0}^{n} p_i - p_i^j = 0$. Now, lets consider the left hand side, which implies

$$\bigwedge_{i=1}^{n} \left(\begin{array}{l} \bigwedge_{l\in 1..m\setminus j} p_i^l \leq 0 \wedge p_i - p_i^1 - \cdots - p_i^m \leq 0 \wedge \\ \bigwedge_{l\in 1..m\setminus j} p_i^{l\prime} \leq 0 \wedge p_i' - p_i^{1\prime} - \cdots - p_i^{m\prime} \leq 0 \wedge \\ (p_i^j + p_i^{j\prime} \leq 0) \wedge -p_i - p_i' \leq 0 \end{array} \right)$$

By adding inequalities of each row, we obtain

$$\bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i - p_i^j \leq 0 \wedge \\ p_i' - p_i^{j\prime} \leq 0 \wedge \\ p_i^j + p_i^{j\prime} - p_i - p_i' \leq 0 \end{array} \right).$$

By adding 2nd and 3rd row, we obtain $\bigwedge_{i=1}^{n}(p_i - p_i^j \leq 0 \wedge p_i^j - p_i \leq 0)$, which we were aiming to prove.

(4) $\mathit{Smb}(f(t_1, \ldots, t_n)) \nsubseteq \mathit{OutSmb}(o) \wedge \mathit{Smb}(f(s_1, \ldots, s_n)) \subseteq \mathit{OutSmb}(o) \wedge$
$\left(\forall j \in 1..m : \begin{array}{l} \mathit{Smb}(f(t_1, \ldots, t_n)) \subseteq \mathit{OutSmb}(o^j) \wedge \\ \mathit{Smb}(f(s_1, \ldots, s_n)) \subseteq \mathit{OutSmb}(o^j) \end{array} \right)$:
Argument is similar to case 2.

(5) $\mathit{Smb}(f(t_1, \ldots, t_n)) \nsubseteq \mathit{OutSmb}(o) \wedge \mathit{Smb}(f(s_1, \ldots, s_n)) \subseteq \mathit{OutSmb}(o) \wedge$
$\left(\exists j \in 1..m : \begin{array}{l} \mathit{Smb}(f(t_1, \ldots, t_n)) \nsubseteq \mathit{OutSmb}(o^j) \wedge \\ \mathit{Smb}(f(s_1, \ldots, s_n)) \subseteq \mathit{OutSmb}(o^j) \end{array} \right)$:
Argument is similar to case 3.

(6) $\mathit{Smb}(f(t_1, \ldots, t_n)) \nsubseteq \mathit{OutSmb}(o) \wedge \mathit{Smb}(f(s_1, \ldots, s_n)) \nsubseteq \mathit{OutSmb}(o) \wedge$
$\left(\forall j \in 1..m : \begin{array}{l} \mathit{Smb}(f(t_1, \ldots, t_n)) \subseteq \mathit{OutSmb}(o^j) \wedge \\ \mathit{Smb}(f(s_1, \ldots, s_n)) \subseteq \mathit{OutSmb}(o^j) \end{array} \right)$:

$C_{r+1} = \mathit{true}$
$D_{r+1} = \bigwedge_{i=1}^{n}(t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p_i' \leq 0)$
$p = f(t_1, \ldots, t_n) - f(s_1, \ldots, s_n)$

For each $l \in 1..m$
$C^l_{r+1} = \bigwedge_{i=1}^{n}(p_i^l \leq 0 \wedge p_i^{l\prime} \leq 0)$
$D^l_{r+1} = \mathit{true}$
$p^l = 0$

(8.7) and (8.8) are trivially true. In the right hand side of (8.9), $p - p^1 - \cdots - p^m = f(t_1, \ldots, t_n) - f(s_1, \ldots, s_n)$. So, we only need to prove that left hand side of (8.9) implies $\bigwedge_{i=1}^{n} t_i = s_i$. By placing values of C^l_{r+1} and D_{r+1}, the left hand side implies

$$\bigwedge_{i=1}^{n}(p_i \leq 0 \wedge p_i' \leq 0 \wedge t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p_i' \leq 0).$$

By taking linear combinations, we obtain $\bigwedge_{i=1}^{n}(t_i - s_i \leq 0 \wedge s_i - t_i \leq 0)$, which we were aiming to prove.

(7) $Smb(f(t_1,\ldots,t_n)) \nsubseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o) \wedge$
$\left(\exists j \in 1..m : \begin{array}{l} Smb(f(t_1,\ldots,t_n)) \nsubseteq OutSmb(o^j) \wedge \\ Smb(f(s_1,\ldots,s_n)) \subseteq OutSmb(o^j) \end{array}\right) \wedge$
$\left(\forall j' \in 1..m \setminus \{j\} : \begin{array}{l} Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o^{j'}) \wedge \\ Smb(f(s_1,\ldots,s_n)) \subseteq OutSmb(o^{j'}) \end{array}\right) :$

$C_{r+1} = true$
$D_{r+1} = \bigwedge_{i=1}^{n}(t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p'_i \leq 0)$
$p = f(t_1,\ldots,t_n) - f(s_1,\ldots,s_n)$

$C^j_{r+1} = \bigwedge_{i=1}^{n}(p^j_i + p^{j\prime}_i \leq 0)$
$D^j_{r+1} = \bigwedge_{i=1}^{n}(-p^j_i - p^{j\prime}_i \leq 0)$
$p^j = f(s_1 + p^j_1,\ldots,s_n + p^j_n) - f(s_1,\ldots,s_n)$

For each $l \in 1..m \setminus \{j\}$
$C^l_{r+1} = \bigwedge_{i=1}^{n}(p^l_i \leq 0 \wedge p^{l\prime}_i \leq 0)$
$D^l_{r+1} = true$
$p^l = 0$

(8.7) is trivially true. For (8.8), we only need to prove the instance of implications in which, D^j_{r+1} is equal to $\bigwedge_{i=1}^{n}(-p^j_i - p^{j\prime}_i \leq 0)$. Lets consider left hand side of (8.8), which is

$$(\textstyle\bigwedge_{l \in 1..m\setminus\{j\}} C^l_{r+1}) \wedge D_{r+1} \;\; \wedge \bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i - p^1_i - \cdots - p^m_i \leq 0 \wedge \\ p'_i - p^{1\prime}_i - \cdots - p^{m\prime}_i \leq 0 \end{array} \right).$$

After placing values of C^l_{r+1} and D_{r+1},

$$\bigwedge_{i=1}^{n} \left(\begin{array}{l} t_i - s_i - p_i \leq 0 \wedge \\ s_i - t_i - p'_i \leq 0 \end{array} \right) \wedge \bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i - p^j_i \leq 0 \wedge \\ p'_i - p^{j\prime}_i \leq 0 \end{array} \right).$$

After adding all inequalities above, we obtain $\bigwedge_{i=1}^{n}\left(-p^j_i - p^{j\prime}_i \leq 0\right)$, which is right hand side of (8.8).

In the right hand side of (8.9), $p - p^1 - \cdots - p^m = f(t_1,\ldots,t_n) - f(s_1 + p^j_1,\ldots,s_n + p^j_n)$. So, we only need to prove that left hand side of (8.9) implies $\bigwedge_{i=1}^{n} t_i = s_i + p^j_i$. By placing values of C^l_{r+1} and D_{r+1}, the left hand side implies

$$\textstyle\bigwedge_{i=1}^{n}(p^j_i + p^{j\prime}_i \leq 0) \;\; \wedge \displaystyle\bigwedge_{i=1}^{n} \left(\begin{array}{l} t_i - s_i - p_i \leq 0 \wedge \\ s_i - t_i - p'_i \leq 0 \end{array} \right) \wedge \bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i - p^j_i \leq 0 \wedge \\ p'_i - p^{j\prime}_i \leq 0 \end{array} \right)$$

by taking linear combination of above equations,

$$\bigwedge_{i=1}^{n} \left(\begin{array}{l} t_i - s_i - p^{j\prime}_i \leq 0 \wedge \\ s_i - t_i - p^j_i \leq 0 \end{array} \right) \wedge \bigwedge_{i=1}^{n} \left(\begin{array}{l} t_i - s_i - p^j_i \leq 0 \wedge \\ s_i - t_i - p^{j\prime}_i \leq 0 \end{array} \right)$$

Therefore, $\bigwedge_{i=1}^{n} t_i = s_i + p^j_i$, which we were aiming to prove.

(8) $Smb(f(t_1,\ldots,t_n)) \nsubseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o) \wedge$
$\left(\exists j \in 1..m : \begin{array}{l} Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o^j) \wedge \\ Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o^j) \end{array}\right) \wedge$
$\left(\forall j' \in 1..m \setminus \{j\} : \begin{array}{l} Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o^{j'}) \wedge \\ Smb(f(s_1,\ldots,s_n)) \subseteq OutSmb(o^{j'}) \end{array}\right)$

A similar argument as in previous case.

(9) $Smb(f(t_1,\ldots,t_n)) \nsubseteq OutSmb(o) \wedge Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o) \wedge$

$$\left(\exists j^1 \in 1..m : \begin{array}{l} Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o^{j^1}) \wedge \\ Smb(f(s_1,\ldots,s_n)) \nsubseteq OutSmb(o^{j^1}) \end{array}\right) \wedge$$
$$\left(\exists j^2 \in 1..m : \begin{array}{l} Smb(f(t_1,\ldots,t_n)) \nsubseteq OutSmb(o^{j^2}) \wedge \\ Smb(f(s_1,\ldots,s_n)) \subseteq OutSmb(o^{j^2}) \end{array}\right) \wedge$$
$$\left(\forall j' \in 1..m \setminus \{j^1,j^2\} : \begin{array}{l} Smb(f(t_1,\ldots,t_n)) \subseteq OutSmb(o^{j'}) \wedge \\ Smb(f(s_1,\ldots,s_n)) \subseteq OutSmb(o^{j'}) \end{array}\right)$$

$C_{r+1} = true$
$D_{r+1} = \bigwedge_{i=1}^n (t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p_i' \leq 0)$
$p = f(t_1,\ldots,t_n) - f(s_1,\ldots,s_n)$

For each $l \in 1..m \setminus \{j^1,j^2\}$
$C^l_{r+1} = \bigwedge_{i=1}^n (p_i^l \leq 0 \wedge p_i^{l\prime} \leq 0)$
$D^l_{r+1} = true$
$p^l = 0$

$C^{j^1}_{r+1} = \bigwedge_{i=1}^n (p_i^{j^1} + p_i^{j^1\prime} \leq 0)$
$D^{j^1}_{r+1} = \bigwedge_{i=1}^n (-p_i^{j^1} - p_i^{j^1\prime} \leq 0)$
$p^{j^1} = f(s_1 + p_1^{j^1},\ldots,s_n + p_n^{j^1}) - f(s_1,\ldots,s_n)$

$C^{j^2}_{r+1} = \bigwedge_{i=1}^n (p_i^{j^2} + p_i^{j^2\prime} \leq 0)$
$D^{j^2}_{r+1} = \bigwedge_{i=1}^n (-p_i^{j^2} - p_i^{j^2\prime} \leq 0)$
$p^{j^2} = f(t_1,\ldots,t_n) - f(t_1 + p_1^{j^2\prime},\ldots,t_n + p_n^{j^2\prime})$

(8.7) is trivially true. In (8.8), there are two non trivial implications, when $j = j^1$ and $j = j^2$. For $j = j^1$, the left hand side of implication is

$$(\textstyle\bigwedge_{l\in 1..m\setminus\{j^1\}} C^l_{r+1}) \wedge D_{r+1} \quad \wedge \displaystyle\bigwedge_{i=1}^n \left(\begin{array}{l} p_i - p_i^1 - \cdots - p_i^m \leq 0 \wedge \\ p_i' - p_i^{1\prime} - \cdots - p_i^{m\prime} \leq 0 \end{array}\right).$$

After placing values of C^l_{r+1} other than $l = j^2$, we obtain

$$C^{j^2}_{r+1} \wedge D_{r+1} \quad \wedge \bigwedge_{i=1}^n \left(\begin{array}{l} p_i - p_i^{j^1} - p_i^{j^2} \leq 0 \wedge \\ p_i' - p_i^{j^1\prime} - p_i^{j^2\prime} \leq 0 \end{array}\right).$$

After placing values of $C^{j^2}_{r+1}$ and D_{r+1}, we obtain

$$\begin{array}{l} \bigwedge_{i=1}^n (p_i^{j^2} + p_i^{j^2\prime} \leq 0) \wedge \bigwedge_{i=1}^n \left(\begin{array}{l} t_i - s_i - p_i \leq 0 \wedge \\ s_i - t_i - p_i' \leq 0 \end{array}\right) \wedge \\ \bigwedge_{i=1}^n \left(\begin{array}{l} p_i - p_i^{j^1} - p_i^{j^2} \leq 0 \wedge \\ p_i' - p_i^{j^1\prime} - p_i^{j^2\prime} \leq 0 \end{array}\right). \end{array}$$

By taking linear combinations, we obtain $\bigwedge_{i=1}^n \left(-p_i^{j^1} - p_i^{j^1\prime} \leq 0\right)$, which is the right hand side. A similar argument proves $j = j^2$ instantiation of (8.8).

The left hand side of (8.9) is

$$\begin{array}{l} (\bigwedge_{l\in 1..m\setminus\{j^1,j^2\}} C^l_{r+1}) \wedge C^{j^1}_{r+1} \wedge C^{j^2}_{r+1} \wedge D_{r+1} \wedge \\ \bigwedge_{i=1}^n \left(\begin{array}{l} p_i - p_i^1 - \cdots - p_i^m \leq 0 \wedge \\ p_i' - p_i^{1\prime} - \cdots - p_i^{m\prime} \leq 0 \end{array}\right). \end{array}$$

By placing values of C^j_{r+1} for $j \in 1..m \setminus \{j^1,j^2\}$, we obtain

$$C^{j^1}_{r+1} \wedge C^{j^2}_{r+1} \wedge D_{r+1} \quad \wedge \bigwedge_{i=1}^n \left(\begin{array}{l} p_i - p_i^{j^1} - p_i^{j^2} \leq 0 \wedge \\ p_i' - p_i^{j^1\prime} - p_i^{j^2\prime} \leq 0 \end{array}\right).$$

After placing value of D_{r+1}, we obtain

$$C^{j^1}_{r+1} \wedge C^{j^2}_{r+1} \wedge \bigwedge_{i=1}^n \left(\begin{array}{l} t_i - s_i - p_i^{j^1} - p_i^{j^2} \leq 0 \wedge \\ s_i - t_i - p_i^{j^1\prime} - p_i^{j^2\prime} \leq 0 \end{array}\right).$$

After placing values of $C_{r+1}^{j^1}$ and $C_{r+1}^{j^2}$, we obtain

$$\bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i^{j^1} + p_i^{j^1\prime} \leq 0 \\ p_i^{j^2} + p_i^{j^2\prime} \leq 0 \end{array} \right) \wedge \bigwedge_{i=1}^{n} \left(\begin{array}{l} t_i - s_i - p_i^{j^1} - p_i^{j^2} \leq 0 \wedge \\ s_i - t_i - p_i^{j^1\prime} - p_i^{j^2\prime} \leq 0 \end{array} \right).$$

By taking linear combinations of above inequalities, we obtain

$$\bigwedge_{i=1}^{n} \left(\begin{array}{l} t_i - s_i + p_i^{j^1\prime} - p_i^{j^2} \leq 0 \wedge \\ s_i - t_i - p_i^{j^1\prime} + p_i^{j^2} \leq 0 \end{array} \right).$$

Therefore,

$$\bigwedge_{i=1}^{n} \left(\; t_i + p_i^{j^1\prime} = s_i + p_i^{j^2} \; \right)$$

Therefore,

$$f(t_1 + p_1^{j^2\prime}, \ldots, t_n + p_n^{j^2\prime}) - f(s_1 + p_1^{j^1}, \ldots, s_n + p_n^{j^1}) \leq 0,$$

which is right hand side of (8.9).

(10) $\mathit{Smb}(f(t_1,\ldots,t_n)) \nsubseteq \mathit{OutSmb}(o) \wedge \mathit{Smb}(f(s_1,\ldots,s_n)) \nsubseteq \mathit{OutSmb}(o) \wedge$
$\left(\exists j \in 1..m : \begin{array}{l} \mathit{Smb}(f(t_1,\ldots,t_n)) \nsubseteq \mathit{OutSmb}(o^j) \wedge \\ \mathit{Smb}(f(s_1,\ldots,s_n)) \nsubseteq \mathit{OutSmb}(o^j) \end{array}\right) \wedge$
$\left(\forall j' \in 1..m \setminus \{j\} : \begin{array}{l} \mathit{Smb}(f(t_1,\ldots,t_n)) \subseteq \mathit{OutSmb}(o^{j'}) \wedge \\ \mathit{Smb}(f(s_1,\ldots,s_n)) \subseteq \mathit{OutSmb}(o^{j'}) \end{array}\right)$:

$C_{r+1} = \mathit{true}$
$D_{r+1} = \bigwedge_{i=1}^{n}(t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p_i' \leq 0)$
$p = f(t_1,\ldots,t_n) - f(s_1,\ldots,s_n)$

$C_{r+1}^{j} = \mathit{true}$
$D_{r+1}^{j} = \bigwedge_{i=1}^{n}(t_i - s_i - p_i^j \leq 0 \wedge s_i - t_i - p_i^{j\prime} \leq 0)$
$p^j = f(t_1,\ldots,t_n) - f(s_1,\ldots,s_n)$

For each $l \in 1..m \setminus \{j\}$
$C_{r+1}^{l} = \bigwedge_{i=1}^{n}(p_i^l \leq 0 \wedge p_i^{l\prime} \leq 0)$
$D_{r+1}^{l} = \mathit{true}$
$p^l = 0$

(8.7) and (8.9) are trivially true. For (8.8), we only need to prove the instance of implications in which, D_{r+1}^{j} is equal to $\bigwedge_{i=1}^{n}(t_i - s_i - p_i^j \leq 0 \wedge s_i - t_i - p_i^{j\prime} \leq 0)$. Lets consider left hand side of (8.8), which implies

$$\bigwedge_{i=1}^{n} \left(\begin{array}{l} t_i - s_i - p_i \leq 0 \wedge \\ s_i - t_i - p_i' \leq 0 \end{array} \right) \wedge \bigwedge_{i=1}^{n} \left(\begin{array}{l} p_i - p_i^j \leq 0 \wedge \\ p_i' - p_i^{j\prime} \leq 0 \end{array} \right).$$

By taking linear combinations, we obtain

$$\bigwedge_{i=1}^{n} \left(\begin{array}{l} t_i - s_i - p_i^j \leq 0 \wedge \\ s_i - t_i - p_i^{j\prime} \leq 0 \end{array} \right),$$

which is the right hand side.

(AI-4): Let $o = \mathit{false}$. The node *false* is root of the resolution tree therefore $\mathit{Smb}(f(t_1,\ldots,t_n)) \nsubseteq \mathit{OutSmb}(o)$ and $\mathit{Smb}(f(s_1,\ldots,s_n)) \nsubseteq \mathit{OutSmb}(o)$. Therefore, $C_{r+1} = \mathit{true}$ and $D_{r+1} = \bigwedge_{i=1}^{n}(t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p_i' \leq 0)$. Since, for each $i \in 1..n$, $p_i = t_i - s_i$ and $p_i' = s_i - t_i$, $D_{r+1} = \mathit{true}$. Hence, (AI-4) w.r.t. $\Pi(o)$ holds. □

Theorem 12 (Complexity)**.** *Application of annotation rules in Figure 8.1 takes linear time in proportion to the size of the proof tree.*

Proof. The annotation of the rules are computed in linear pass by depth first traversal of a proof-tree. □

```
// take_lock : multi-thread program
int f[N];
int p, q;

// Thread1(int c)                        // Thread2(int d)
a1:  assume(p <= c <= q);                b1:  assume(q <= d <= p);
a2:  take_lock(f, c);                    b2:  take_lock(f, d);
a3:  // critical                         b3:  // critical
```

(a)

```
int p, q;

int main() {                             void foo() {
m1:   int c = ..;                        n1:   int d = ..;
m2:   assume(p <= c <= q);               n2:   assume(q <= d <= p);
m3:   if (f(c) == 1) { foo(); }          n3:   if (f(d) == 0)
m4:   assert(false);                     n4:     return;
}                                        n5:   ...
                                         }
```

(b)

Figure 8.4: Two example programs take_lock and main. (a) take_lock illustrates how Horn clauses can represent an abstraction refinement task in presence thread interaction. (b) main illustrates a formalization the abstraction refinement for programs with procedures using Horn clauses.

8.4 Illustration: obtaining Horn clauses from refinement

This section presents examples of Horn clauses obtained during the abstraction refinement step when verifying multi-threaded programs and programs with procedures.

Abstraction refinement for multi-threaded programs

See Figure 8.4(a) for a program take_lock that consists of two threads. These threads attempt to access a critical section and synchronize their accesses using a lock stored in the global array f. The two threads receive the identifier of the lock as an integer argument c for the first thread and d for the second thread. The assume statements at labels a1 and b1 ensure that the two integer indices, c and d , are equal. The calls at labels a2 and b2 ensure that the two threads cannot both enter the critical section, i.e., the assertion $\neg(\mathtt{pc}_1 = \mathtt{a3} \wedge \mathtt{pc}_2 = \mathtt{b3})$ holds for all executions of the program. We write $V = \{\mathtt{f}, \mathtt{p}, \mathtt{q}, \mathtt{c}, \mathtt{d}, \mathtt{pc}_1, \mathtt{pc}_2\}$ for the set of all program variables, where $\mathtt{pc}_1$ and $\mathtt{pc}_2$ are local program counter variables of the first and second thread, respectively. Let $G = \{\mathtt{p}, \mathtt{q}\}$ be the set of global program variables.

To verify the program take_lock, the method described in [37] performs abstract reachability computations for each thread considering both local thread transitions and environment transitions that capture updates of program state done by the other thread. Let us assume that the abstract reachability procedure finds a spurious error state following an interleaving of the statements from the two threads represented by two assertions ρ_1 and ρ_2 .

The results computed by the abstract reachability are an abstract state s and an environment transition e such that:

$$s = \dot{\alpha}(post(\rho_1, true)),$$
$$e = \ddot{\alpha}(\rho_2),$$

where *post* denotes the successor function and by $\dot{\alpha}$ and $\ddot{\alpha}$ denote abstraction functions for over-approximation of sets of states and sets of pairs of states, respectively. The constraint ρ_1 represents program statements at location `a1` and `a2` from the first thread, while ρ_2 represents the program statements at locating `b1` and `b2` from the second thread. Both transitions are over unprimed and primed program variables. We only show the critical part of these constraints that is relevant to the infeasibility of the interleaving:

$$\rho_1 = (\mathtt{p} \le \mathtt{c} \wedge \mathtt{c} \le \mathtt{q} \wedge \mathtt{f}(\mathtt{c}) = 1 \wedge \mathtt{p} = \mathtt{p}' \wedge \mathtt{q} = \mathtt{q}' \wedge \mathtt{c} = \mathtt{c}')\ ,$$
$$\rho_2 = (\mathtt{q} \le \mathtt{d} \wedge \mathtt{d} \le \mathtt{p} \wedge \mathtt{f}(\mathtt{d}) = 0 \wedge \mathtt{p} = \mathtt{p}' \wedge \mathtt{q} = \mathtt{q}' \wedge \mathtt{d} = \mathtt{d}')\ .$$

We model the fact that the first thread acquires the lock indexed by `c` using $\mathtt{f}(\mathtt{c}) = 1$. The constraint $\mathtt{f}(\mathtt{d}) = 0$ from ρ_2 represents the requirement that the lock indexed by `d` must be released in order to complete the call to `take_lock` at program location `b2` .

Following the reachability of an abstract state that intersects the error states $(\mathtt{pc}_1 = \mathtt{a3} \wedge \mathtt{pc}_2 = \mathtt{b3})$, an abstraction refinement constraints are derived. We obtain a set of Horn clauses where the unknown query $S(V)$ represents the refined abstract state s and $E(G, G')$ represents the refined environment transition e :

$$\mathcal{HC}_{\mathtt{take\_lock}} = \{\ \rho_1 \to S(V'),\ \rho_2 \to E(G, G'),\ S(V) \wedge E(G, G') \to \mathit{false}\ \}\ .$$

The third clause requires that the intersection of the set of states $S(V)$ and the environment transition is empty. While solutions for the refined environment transitions can be expressed in terms of the whole set of program variables V , an efficient verification procedure relies on using thread-modular solutions whenever they exist. In particular, we consider in our example $E(G, G')$.

Each Horn clause is implicitly universally quantified over the variables that appear in the clause, i.e., V and V'. The set of clauses $\mathcal{HC}_{\mathtt{take\_lock}}$ is satisfiable if and only if the abstraction can be refined to exclude the spurious interleaving.

Abstraction refinement for programs with procedures

We use the second program in Figure 8.5 to illustrate refinement constraints for proving the infeasibility of an interprocedural path that is expressed using Horn clauses. This program has the same set of program variables V and program global variables G as `take_lock`.

The procedure `main` establishes at line `m2` that the value of the local variable `c` is in a required range of integer values. At line `m3` , `foo` is called if an unspecified function `f` returns the integer value 1. Due to the conditions at lines `n2` and `n3` , the procedure `foo` cannot return at line `n4` from the calling context at line `m3`. However, due to over-approximation, an abstract reachability computation may result in a summary for the `foo` procedure that is too imprecise. Assuming that the constraint ρ_1 represents the calling context of `foo` at line `m3`.

$$\rho_1 \quad = \quad (\mathtt{p} \le \mathtt{c} \wedge \mathtt{c} \le \mathtt{q} \wedge \mathtt{f}(\mathtt{c}) = 1 \wedge \mathtt{p} = \mathtt{p}' \wedge \mathtt{q} = \mathtt{q}' \wedge \mathtt{c} = \mathtt{c}')\ ,$$

An abstract state s is computed as follows:

$$s \quad = \quad \dot{\alpha}(\mathit{post}(\rho_1, \mathit{true}))\ .$$

Further, using a transition abstraction function $\ddot{\alpha}$, a summary transition e is computed for the `foo` procedure:

$$\rho_2 \quad = \quad (\mathtt{q} \le \mathtt{d} \wedge \mathtt{d} \le \mathtt{p} \wedge \mathtt{f}(\mathtt{d}) = 0 \wedge \mathtt{p} = \mathtt{p}' \wedge \mathtt{q} = \mathtt{q}')\ ,$$
$$e \quad = \quad \ddot{\alpha}(\rho_2)\ .$$

In order to show the infeasibility of the interprocedural path denoted by the sequence of program labels $\mathtt{m1}, \mathtt{m2}, \mathtt{m3}, \mathtt{n1}, \mathtt{n2}, \mathtt{n3}, \mathtt{n4}, \mathtt{m4}$, abstraction refinement constraints are expressed by the following Horn clauses:

$$\mathcal{HC}_{\mathtt{foo}} = \{\ \rho_1 \to S(V'),\ \rho_2 \to E(G, G'),\ S(V) \wedge E(G, G') \to \mathit{false}\ \}\ .$$

We require that the solution for the procedure summary refers only to global variables `p` and `q`, but not to the local variable `d`. Therefore, $E(G, G')$ refers to only global variables.

8.5 Illustration: solving Horn clauses

We constructed the above examples such that $\mathcal{HC}_{\texttt{take\_lock}} = \mathcal{HC}_{\texttt{foo}}$. We further simplify the Horn clauses and drop the variables from the queries that do not contribute to the satisfiablity of the set of Horn clauses. After the simplification, we obtain

$$\mathcal{HC} = \{\ \rho_1 \rightarrow S(\mathtt{p}, \mathtt{q}, \mathtt{c}),\ \rho_2 \rightarrow E(\mathtt{p}, \mathtt{q}),\ S(\mathtt{p}, \mathtt{q}, \mathtt{c}) \wedge E(\mathtt{p}, \mathtt{q}) \rightarrow \mathit{false}\ \}\ .$$

In Figure 8.5(a), expanded version of $\mathcal{HC}$ is presented and, since our algorithm we assume that no two Horn clauses share a variable, we have added a different subscripts in variables in each Horn clause. This section illustrates how our Horn clauses solving algorithm applies to $\mathcal{HC}$.

Resolution tree

Our solving algorithm starts by constructing from $\mathcal{HC}$ a resolution tree R shown in Figure 8.5(b). We label nodes of R with indices for easy reference. The root of the tree is labeled with the atom *false*, prefixed by an index 1 used to refer to the node. For each clause from $\mathcal{HC}$, we add edges to R between the node corresponding to the head of the clause and the nodes corresponding to the body of the clause. For example, the first clause leads to an edge between the node 2 corresponding to the head $S(\mathtt{p}, \mathtt{q}, \mathtt{c})$ and the nodes labeled 3–6 corresponding to a conjunction of atomic predicates from the body of the same clause.

Proof tree

Next, our algorithm constructs a proof tree that proves unsatisfiability of the constraints from the leaves of the resolution tree, using proof rules presented in Section 2.2. The resulting proof tree P is shown in Figure 8.5(c). The linear combination rule PComb is applied to derive the constraint $(\mathtt{c} - \mathtt{d} \leq 0)$ from the premises $(\mathtt{c} - \mathtt{q} \leq 0)$ and $(\mathtt{q} - \mathtt{d} \leq 0)$. PComb is also used to derive $(\mathtt{d} - \mathtt{c} \leq 0)$ from the premises $(\mathtt{p} - \mathtt{c} \leq 0)$ and $(\mathtt{d} - \mathtt{p} \leq 0)$. The congruence rule PCong derives $(\mathtt{f}(\mathtt{c}) - \mathtt{f}(\mathtt{d}) \leq 0)$ from the premises $(\mathtt{c} - \mathtt{d} \leq 0)$ and $(\mathtt{d} - \mathtt{c} \leq 0)$. Lastly, $(1 \leq 0)$ is derived by applying PComb on three premises, $(\mathtt{f}(\mathtt{d}) \leq 0)$, $(\mathtt{f}(\mathtt{c}) - \mathtt{f}(\mathtt{d}) \leq 0)$, and $(-\mathtt{f}(\mathtt{c}) + 1 \leq 0)$.

Annotated trees and solution

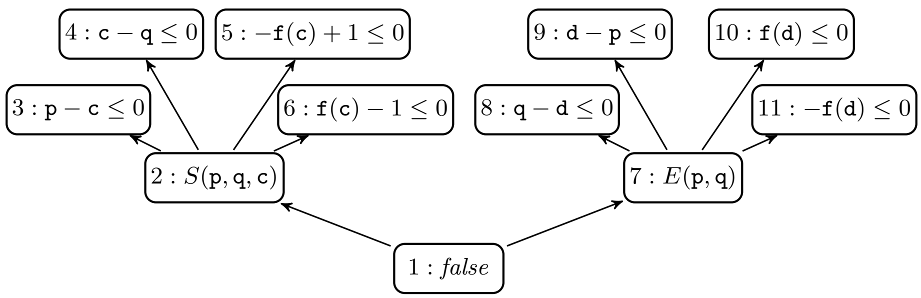
For each node in P, our algorithm creates an annotated version of R. These annotation trees are partial solutions, as we have discussed earlier. Figure 8.5(d) presents a part of the annotated proof tree and Figure 8.6 presented expanded view of some of the annotations. In P, $\mathtt{c} - \mathtt{d} \leq 0$ is derived by adding $(\mathtt{c} - \mathtt{q} \leq 0)$ and $(\mathtt{q} - \mathtt{d} \leq 0)$. Therefore, annotations Π_3 for node $\mathtt{c} - \mathtt{d} \leq 0$ is result of procedure call $\textsc{SolComb}(\Pi_1, \Pi_2, 1, 1)$.

Following the derivation of the proof tree P, annotation rules are used to combine annotated trees until those corresponding to the rule applied at the bottom of the proof tree. Π from Figure 8.6 shows the final solution computed by the last application of an inference rule. The node labeled "2" contains the solution for $S(\mathtt{p}, \mathtt{q}, \mathtt{c})$ and it can be simplified to $S(\mathtt{p}, \mathtt{q}, \mathtt{c}) = (\mathtt{p} < \mathtt{q} \vee \mathtt{p} \leq \mathtt{q} \wedge \mathtt{f}(\mathtt{p}) \geq 1)$. The solution from the node labeled "7" can be simplified to $E(\mathtt{p}, \mathtt{q}) = (\mathtt{p} > \mathtt{q} \vee \mathtt{p} \geq \mathtt{q} \wedge \mathtt{f}(\mathtt{p}) \leq 0)$.

The existence of a solution for the set of Horn clauses $\mathcal{HC}$ indicates that the counterexamples discovered for the programs `take_lock` and `foo` are spurious. Refining the abstraction with the atomic predicates that appear in the solutions of $S(\mathtt{p}, \mathtt{q}, \mathtt{c})$ and $E(\mathtt{p}, \mathtt{q})$ guarantees that the same spurious counterexample will not appear during subsequent abstract reachability computations.

$$\begin{aligned}\mathcal{HC} = \{\ & \texttt{p}_1 \le \texttt{c}_1 \land \texttt{c}_1 \le \texttt{q}_1 \land -\texttt{f}(\texttt{c}_1) + 1 \le 0 \land \texttt{f}(\texttt{c}_1) - 1 \le 0 \to S(\texttt{p}_1, \texttt{q}_1, \texttt{c}_1),\\ & \texttt{q}_2 \le \texttt{d}_2 \land \texttt{d}_2 \le \texttt{p}_2 \land \texttt{f}(\texttt{d}_2) \le 0 \land -\texttt{f}(\texttt{d}_2) \le 0 \to E(\texttt{p}_2, \texttt{q}_2),\\ & S(\texttt{p}_3, \texttt{q}_3, \texttt{c}_3) \land E(\texttt{p}_3, \texttt{q}_3) \to \mathit{false}\ \}\end{aligned}$$

(a)



(b)

$$\textsc{PComb}\ \dfrac{\textsc{PHyp}\ \dfrac{}{\texttt{f}(\texttt{d}) \le 0} \quad \textsc{PCong}\ \dfrac{\dfrac{\texttt{c} - \texttt{q} \le 0 \quad \texttt{q} - \texttt{d} \le 0}{\texttt{c} - \texttt{d} \le 0} \quad \dfrac{\texttt{p} - \texttt{c} \le 0 \quad \texttt{d} - \texttt{p} \le 0}{\texttt{d} - \texttt{c} \le 0}}{\texttt{f}(\texttt{c}) - \texttt{f}(\texttt{d}) \le 0} \quad \textsc{PHyp}\ \dfrac{}{-\texttt{f}(\texttt{c}) + 1 \le 0}}{1 \le 0}$$

(c)

$$\textsc{PComb}\ \dfrac{\textsc{PHyp}\ \dfrac{}{\texttt{f}(\texttt{d}) \le 0[\dots]} \quad \textsc{PComb}\ \dfrac{\dfrac{\texttt{c} - \texttt{q} \le 0[\Pi_1] \quad \texttt{q} - \texttt{d} \le 0[\Pi_2]}{\texttt{c} - \texttt{d} \le 0[\Pi_3]} \quad \dots}{\texttt{f}(\texttt{c}) - \texttt{f}(\texttt{d}) \le 0[\dots]} \quad \dots}{1 \le 0[\Pi]}$$

(d)

Figure 8.5: (a) A set of Horn clauses $\mathcal{HC}$. (b) Corresponding resolution tree R . (c) Proof of unsatisfiability P for the constraints from the leaves of the resolution tree. For abbreviation, we did not mark nodes of subtree of $\texttt{f}(\texttt{c}) - \texttt{f}(\texttt{d}) \le 0$ with the applied proof rules. (d) A part of the annotated proof tree. The annotations Π_1, Π_2,Π_3, and Π are presented in Figure 8.6.

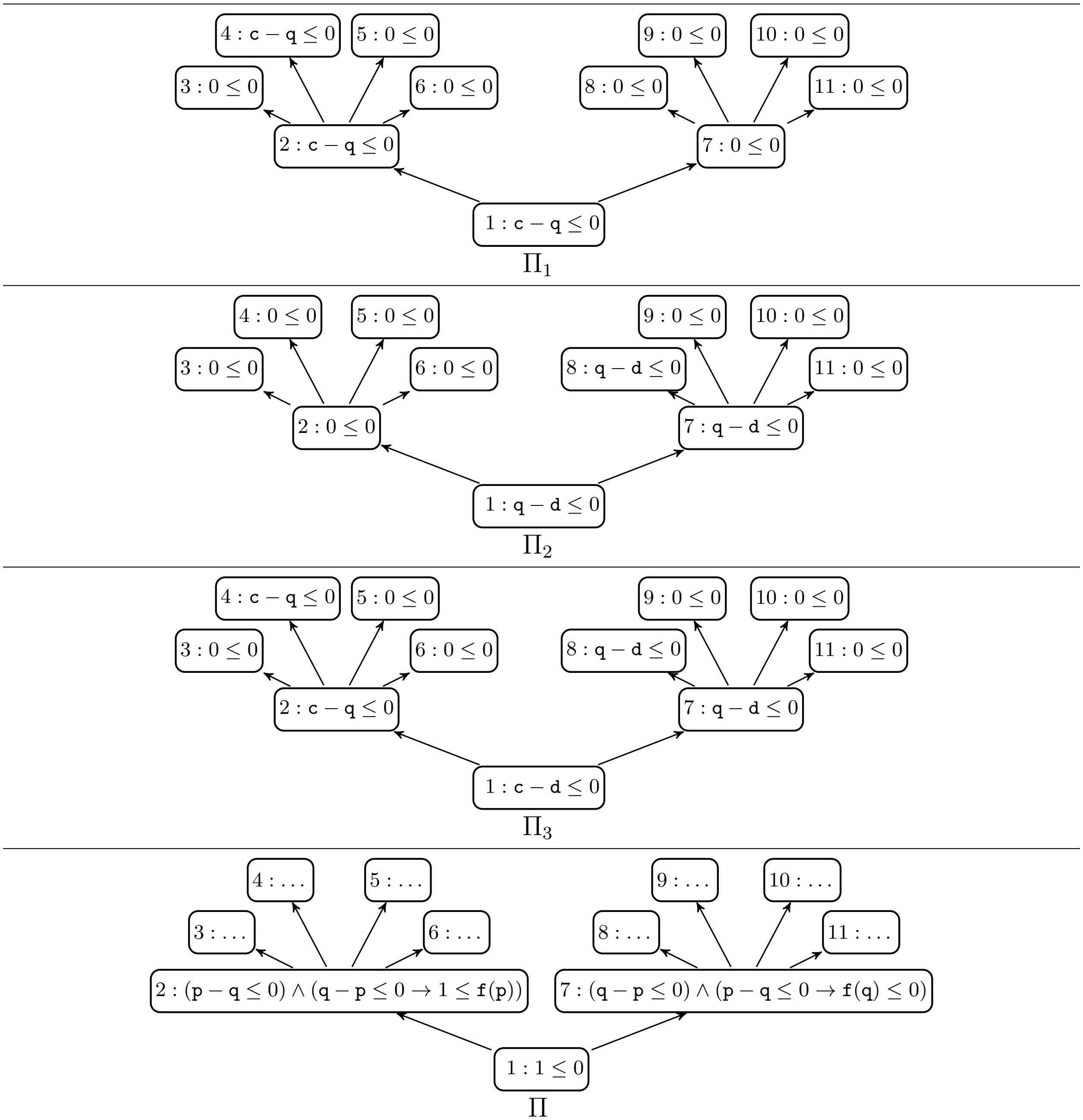


Figure 8.6: Four annotated trees Π_1 , Π_2 , Π_3 , and Π . Π_1 and Π_2 are annotations of nodes $(\mathtt{c} - \mathtt{q} \leq 0)$ and $(\mathtt{q} - \mathtt{d} \leq 0)$ in P, respectively. Π_3 is obtained by applying the combination rule HcComb to Π_1 and Π_2. Π annotates $1 \leq 0$ in P. Therefore, the final solution of $\mathcal{HC}$ can be derived from Π: the node labeled "2" contains the solution for $S(\mathtt{p}, \mathtt{q}, \mathtt{c})$, while the node labeled "7" contains the solution for $E(\mathtt{p}, \mathtt{q})$.

Bibliography

[1] E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM*, 2007.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[3] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, 2009.

[4] D. Beyer, T. Henzinger, R. Majumdar, and A.Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, 2007.

[5] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, 2007.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ICFP*, 1998.

[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.

[8] V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 2006.

[9] B. C. Brock, W. A. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. *Technical Report 86,* `http://www.cli.com`, 1994.

[10] F. Bruschi and F. Ferrandi. Synthesis of complex control structures from behavioral SystemC models. *DATE*, 2003.

[11] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The mathsat 4smt solver. In *CAV*, 2008.

[12] B. A. Buyukkurt, Z. Guo, and W. Najjar. Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs. *Int. Workshop On Applied Reconfigurable Computing*, 2006.

[13] B. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.

[14] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *TACAS*, pages 397–412, 2008.

[15] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Logic*, 12, November 2010.

[16] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.

[17] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD*, 2009.

[18] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, 2005.

[19] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78*, pages 84–96. ACM Press, 1978.

[20] W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic*, 22(3):pp. 250–268, 1957.

[21] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[22] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. *TACAS*, 2006.

[23] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.

[24] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):1–25, 2001.

[25] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.

[26] J. Farkas. Uber die theorie der einfachen ungleichungen. *Journal fur die Reine und Angewandte Mathematik*, 124:1–27, 1902.

[27] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, page 1, 1967.

[28] P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[29] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223. ACM, 2005.

[30] M. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. *FCCM*, 2000.

[31] L. Gonnord and N. Halbwachs. Combining widening and acceleration in Linear Relation Analysis. In *Proc. SAS*, LNCS 4134, pages 144–160. Springer, 2006.

[32] D. Gopan and T. Reps. Lookahead widening. In *Proc. CAV*, LNCS 4144, pages 452–466. Springer, 2006.

[33] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS*, 2008.

[34] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[35] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.

[36] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *TACAS*, 2009.

[37] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011.

[38] A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *APLAS*, 2011.

[39] A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV*, 2011.

[40] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, 2009.

[41] A. Gupta and A. Rybalchenko. InvGen: an efficient invariant generator, 2009. `http://www7.in.tum.de/ guptaa/invgen/`.

[42] R. K. Gupta and S. Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, 14, Apr. 1997.

[43] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. *VLSI Conference*, 2003.

[44] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.

[45] T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST software verification system. In *SPIN*, pages 25–26, 2005.

[46] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.

[47] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[48] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.

[49] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, 2003.

[50] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *ESOP*, 2006.

[51] C. Holzbaur. A specialized, incremental solved form algorithm for systems of linear inequalities. Technical report, Austrian Research Institute for Artificial Intelligence, 1994.

[52] C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

[53] J.-L. Imbert, J. Cohen, and M.-D. Weeger. An algorithm for linear constraint solving: its incorporation in a prolog meta-interpreter for clp. *The Journal of Logic Programming*, 16(3-4):235 – 253, 1993.

[54] IMEC. CleanC analysis tools. `http://www.imec.be/CleanC/`, 2008.

[55] R. Iosif, M. Bozga, A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, 2006.

[56] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV*, LNCS 4144, pages 137–151. Springer, 2006.

[57] R. Jhala and R. Majumdar. Counterexample refinement for functional programs. available from http://www.cs.ucla.edu/˜rupak/Papers/CEGARFunctional.ps, 2009.

[58] D. Kapur. Automatically generating loop invariants using quantifier elimination. Technical Report 05431 (*Deduction and Applications*), IBFI Schloss Dagstuhl, 2006.

[59] K. Ku, T. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *ASE*, 2007.

[60] G. Lalire, M. Argoud, and B. Jeannet. The interproc analyzer, 2009. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`.

[61] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. *SAS*, 2000.

[62] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. *SAS*, 2006.

[63] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV*, pages 428–432, 2008.

[64] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety.* Springer-Verlag, 1995.

[65] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[66] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[67] A. Miné. The octagon abstract domain. *Higher-Order and Symb. Comp.*, 19:31–100, 2006.

[68] W. A. Najjar, A. P. W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8), 2003.

[69] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.

[70] A. Orden. On the solution of linear equation/inequality systems. *Mathematical Programming*, 1:137–152, 1971.

[71] A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL*, 2007.

[72] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, pages 346–362, 2007.

[73] X. Saint-Mleux, M. Feeley, and J.-P. David. SHard: a Scheme to hardware compiler. In *Workshop on Scheme and Functional Programming*, 2006.

[74] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *Proc. SAS*, LNCS 3148, pages 53–68. Springer, 2004.

[75] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. POPL*, pages 318–329. ACM, 2004.

[76] S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using Mathematical Programming. In *VMCAI*, 2005.

[77] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[78] L. Semeria and G. D. Micheli. SpC: synthesis of pointers in C. *ICCAD*, 1998.

[79] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for C. In *Proc. FSE*, pages 263–272. ACM, 2005.

[80] A. Takach, B. Bower, and T. Bollaert. C based hardware design for wireless applications. *DATE*, 2005.

[81] T. Terauchi. Dependent types from counterexamples. In *POPL*, 2010.

[82] The Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 2001. Release 3.8.7.

[83] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP*, 2009.

[84] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis. DWARV: Delftworkbench automated reconfigurable VHDL generator. *FPL*, 2007.

[85] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In *CADE*, pages 353–368, 2005.