ABSTRACT

Title of dissertation: SATISFIABILITY-BASED PROGRAM REASONING AND PROGRAM SYNTHESIS

Saurabh Srivastava, Doctor of Philosophy, 2010

Dissertation directed by: Professor Jeffrey S. Foster
Department of Computer Science

Program reasoning consists of the tasks of automatically and statically verifying correctness and inferring properties of programs. Program synthesis is the task of automatically generating programs. Both program reasoning and synthesis are theoretically undecidable, but the results in this dissertation show that they are practically tractable. We show that there is enough structure in programs written by human developers to make program reasoning feasible, and additionally we can leverage program reasoning technology for automatic program synthesis.

This dissertation describes expressive and efficient techniques for program reasoning and program synthesis. Our techniques work by encoding the underlying inference tasks as solutions to satisfiability instances. A core ingredient in the reduction of these problems to finite satisfiability instances is the assumption of templates. Templates are user-provided hints about the structural form of the desired artifact, e.g., invariant, pre- and postcondition templates for reasoning; or program templates for synthesis. We propose novel algorithms, parameterized by suitable templates, that reduce the inference of these artifacts to satisfiability.

We show that fixed-point computation—the key technical challenge in program reasoning—is encodable as SAT instances. We also show that program synthesis can be viewed as generalized verification, facilitating the use of program reasoning tools as synthesizers. Lastly, we show that program reasoning tools augmented with symbolic testing can be used to build powerful synthesizers with approximate guarantees.

We implemented the techniques developed in this dissertation in the form of the VS^3—Verification and Synthesis using SMT Solvers—suite of tools. Using the VS^3 tools, we were able to verify and infer expressive properties of programs, and synthesize difficult benchmarks from specifications. These prototype tools demonstrate that we can exploit the engineering advances in current SAT/SMT solvers to do automatic program reasoning and synthesis. We propose building future automatic program reasoning and synthesis tools based on the ideas presented in this dissertation.

SATISFIABILITY-BASED PROGRAM REASONING AND PROGRAM SYNTHESIS

by

Saurabh Srivastava

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:
Professor Jeffrey S. Foster, Chair/Advisor
Professor Michael W. Hicks
Dr. Sumit Gulwani
Professor Jonathan Katz
Professor Mike Boyle

To my parents,
Preeti and Prakash,
and my grandparents.

Acknowledgments

I am greatly indebted to my advisor, Jeff Foster, whose willingness to accept, encourage, and fund, every experimental and contentious research thought I decided to pursue, amazes me in hindsight. It would be fair to say that he introduced me to research in programming languages. My interest in the foundations of programming languages started with a course he taught. To say that the knowledge I got from him was just technical would be a gross understatement. Research is half about knowing what problems to solve. Jeff has given me the perspective to pursue the right research ideas, and also the right attitude to overcome the hurdles a research project invariably presents. Not only that, his constant advising on matter not just technical, has helped hone my skills in writing, presentation, articulation, and maybe even helped me become a more capable social being.

Talking of the research and mentoring "process," there is probably no better person to refer to than Mike Hicks, my second advisor. Mike has the uncanny ability to recognize a student's aptitude, interests, and potential, and then to encourage them in exactly the right way. I started working in programming languages as his student and he introduced me to the rigor in the foundations of programming languages—which he knew would attract me to the field. His own research diversity and time management skills have been a constant source of inspiration and I hope I can one day be as efficient a researcher as he is. His advising and presence ensured that my graduate career was a breeze and enjoyable every step of the way.

"This appears to be an impossible problem. Let us see if we can solve it." If this was ever a research philosophy, Sumit Gulwani's would be this. From Sumit, my third advisor, I have learned that a research agenda is not useful if it is not adventurous, ambitious, and borderline crazy. In my internship under his guidance, Sumit introduced me to program verification, and we decided to ignore traditional approaches in favor of an experimental satisfiability-based approach to program reasoning. Our experimental program reasoning approach served as a segue into automatic program synthesis, a problem typically considered intractable. Also, under his mentorship, I have learnt that research is a social, collaborative, activity. Only through constant conversation do guesses and intuition develop into concrete solutions. Lastly, from his work ethic I have learned that a task worth pursuing is a task worth doing well; even if that means spending 16 to 18 hours a day on it.

I have gained immensely from my interactions with all three of my advisors. I persevere to constantly change myself to imbibe the qualities they each possess, and I admire. I am grateful to them for having invested as much time as they did, and for showing me the way forward.

My graduate research career has been meandering, and I would not have been in my current position, if it had not been for the efforts of my advisors "on the way". A special thanks goes out to Bobby Bhattacharjee, whose prodding is partly the reason why I stuck around in graduate school. He has mentored me in all things non-technical, and indoctrinated in me the perspective that the only livable place is academia! His motivation and encouragement has helped me immensely.

Also, a thanks to my undergraduate research advisors, Dheeraj Sanghi and Ajit K. Chaturvedi, at IIT Kanpur, who got me started on research early.

I would also like to thank other professors in the department and on my committee. I am grateful to Samir Khuller, Aravind Srinivasan, Jonathan Katz, and Mike Boyle for having taken the time out to give me very useful feedback on my research.

A shout-out is in order to the graduate students, present and past, that make up the Programming Languages group at the University of Maryland (PLUM). Nik, Polyvios, Iulian, Chris, Yit, Elnatan, Mark, David An, Mike Furr, Martin, Evan, Avik, Stephen, Ted, and David Greenfieldboyce were all a constant source of enlightening conversations (research or otherwise). Also, to all my other friends in the department, Srinivas Kashyap, Shiv, Bhargav, Aswin, Gaurav, Narayanan, Akhil, and Abheek; who made my graduate school days worth every moment.

Only half jokingly, a nod to all the local coffee shops, without their presence there would be no caffeine induced writing rampages and this dissertation would have taken twice as long.

A special thanks to my family, immediate and extended. I am humbled by the constant support and encouragement that my parents have provided throughout this period. My father, a professor and researcher in Inorganic Chemistry, has been a constant role model for my professional life; while my mother, with her open-mindedness and enthusiasm for absorbing ideas has been a constant role model for my personal life. I am also thankful to them for providing the most intellectually nurturing environment that a child could ask for, and for setting the right expectations for me as an adult.

Also, a big thanks to my younger brothers, Abhishek and Prashast. They have been a source of constant pride and perspective. They are my connection to the non-academic world. Lastly, to the person whose constant presence made the last four years incredibly enjoyable, and who I cannot thank enough, Lauren.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

SAT	Propositional Satisfiability
SMT	Satisfiability Modulo Theories
VS^3	Verification and Synthesis using SMT Solvers
PINS	Path-based Inductive Synthesis
SPANS	Satisfiability-based Program Analysis and Synthesis

Chapter 1

Introduction

> *"If I have a thousand ideas and only one turns out to be good, I am satisfied."*
>
> — Alfred Bernhard Nobel[1]

We invest lots of time and money in software development, and despite major advances in software engineering practice, software development is still tedious, costly, and error-prone. Despite building software being inefficient, more and more of our personal devices are leveraging the flexibility that software provides, and software is increasingly being used to control critical systems; such as automotive and flight control, and financial and medical services. Hence, there is an increasing need to build certifiably correct software, and to do it in a cost-efficient way.

This dissertation addresses two aspects of this problem: program reasoning and program synthesis. Program reasoning consists of proof inference (verification) and specification inference, and program synthesis consists of program inference. Verification is the task of proving that a program meets its specification. Specification inference is the task of inferring properties that hold of a given program.

[1] Swedish Chemist, Engineer and Inventor of dynamite, who used his enormous fortune to institute the Nobel Prizes. 1833-1896. In context, the idea in this dissertation will be to generate constraints, for which if a solver finds *any good* solution that is *satisfying*, then that correspond to solutions to the original programming language problem.

Program synthesis is the task of inferring a program that matches a given specification.

There has been a lot of work on program reasoning and less so on synthesis. Despite significant work on formal methods [164, 72, 98], tools for reasoning about software programs are not commonplace. This is partly because of the inability of currents tools to automatically infer formal descriptions of commonly occurring program constructs, e.g., formulae that quantify over all elements of a data structure. We need to develop techniques that can infer arbitrarily expressive formulae, required for program reasoning in practice. We find that enabling inference of expressive properties will also enable automatic program synthesis. In fact, we show that program reasoning tools can be used to directly build program synthesizers. However, the lack of expressivity in current tools is not surprising, as even *checking* formulae in the presence of quantification is theoretically undecidable. In this dissertation, we show how with minimal help from the user we can build techniques that *infer* arbitrarily expressive program properties, and indeed also synthesize programs.

The thesis we explore in this dissertation is the following: *We can build expressive and efficient techniques for program reasoning and program synthesis by encoding the underlying inference tasks as solutions to satisfiability instances.*

The key technical tools we apply towards this thesis are solvers for satisfiability. Significant engineering effort has led to powerful solvers for propositional satisfiability (SAT) and satisfiability modulo theories (SMT). However, program reasoning and synthesis are not directly encodable as SAT or SMT instances. Therefore, we have to develop the theoretical underpinnings of a satisfiability-based approach to program reasoning and synthesis. While SAT/SMT solvers have previously been used to validate guesses about program properties [17, 255, 15, 3], we instead encode the program property (for reasoning) and even the program (for synthesis) as models of a satisfiability instances. This finite encoding is facilitated by hints provided by the user. Thus solving the satisfiability instance directly solves the programming languages problem.

1.1 Satisfiability- and Template-based Program Reasoning and Synthesis

Propositional satisfiability, specifically 3SAT, is arguably the most studied NP-complete problem. Propositional satisfiability is the problem of finding a boolean assignment to the atomic boolean variables in a formula such that the formula evaluates to `true`. The 3SAT version, in which the formulae are in CNF form with at least 3 disjuncts in each clause, is NP-Complete. Satisfiability modulo theories (SMT) addresses the satisfiability problem in which the atoms are facts from particular theories instead of propositional variables. So, $(b_1 \vee b_2 \vee b_3) \wedge (b_1 \vee b_5 \vee b_6)$ is an example SAT formula, while $(x = y \vee x > z \vee y < z) \wedge (x = y \vee x > z - 10 \vee y < z)$ is an example SMT formula with atoms from the theory of linear arithmetic.

While 3SAT is NP-complete, in recent years researchers have developed many tools that can efficiently solve even very large SAT instances arising in practice. Even further, due to the development of fast decision procedures for particular theories, and their integration into the core SAT solving techniques, has resulted in SMT solvers that are capable of solving large SMT instances, from domains such as hardware and software verification [20]. These tools can solve difficult benchmarks from program verification in the order of a couple of seconds [18].

In this dissertation, we apply SAT and SMT solvers to problems they have not been used in before, e.g., invariant and pre-/postcondition *inference* and program *synthesis*. While they have been engineered to be fast on verification benchmarks where the proof of correctness is provided by the user, our experiments in this dissertation show that the solvers are also efficient on instances arising out of proof inference, i.e., for program reasoning, and program inference, i.e., for program synthesis.

1.1.1 Satisfiability for Reasoning

Webster's dictionary defines "reasoning" as inference of a statement offered in explanation or justification. Our view of reasoning about programs consists of offering justifications for specific properties such as correctness or termination, i.e., verification, and inferring descriptions of their input-output characteristics and the associated justification for why the properties hold, i.e., specification inference. These formal justifications come in the form of *program invariants* that we infer. Invariants are tricky to infer for loops.

The key difficulty in automatic program verification is in inferring *inductive loop invariants*. We treat specification inference as an extension of the verification problem in which one infers invariants about the pre- or postcondition in addition to inferring loop invariants. We desire that the facts we infer about the precondition be the weakest possible and the postcondition be the strongest possible. Inferring weakest preconditions ensures that any other valid precondition is a specialization of the inferred precondition. Analogously, inferring the strongest postcondition ensures that any other valid postcondition is a specialization of the inferred postcondition.

Background: The difficulty in program reasoning The key difficulty in automatic program reasoning is the task of inferring suitable invariants. At a particular program location an assertion over the program state is an invariant if it always holds whenever control reaches that location. A program state, σ, is a mapping of program variables to values, e.g., $\sigma_0 = \{x \mapsto 0, y \mapsto 2, k \mapsto 0\}$ is a state that maps the program variables x, y and k to $0, 2$ and 0, respectively. An assertion holds in a state σ, if the assertion evaluated at the program state is `true`. For example $x = 2k|_{\sigma_0}$ evaluates to `true`, where $p|_\sigma$ is notation for evaluating a predicate p under the map σ.

Loop invariants are assertions at loop header locations, i.e., invariants that hold when entering a loop and in each iteration through the loop. A loop invariant is inductive if it can be shown to hold after an iteration assuming it holds at the beginning of the iteration.

Example 1.1 *Given the following program:*

$$\texttt{x} := \texttt{0}; \texttt{k} := \texttt{0}; \texttt{y} := \texttt{2}; \texttt{while}(*)\{\texttt{x} := \texttt{x} + \texttt{y}; \texttt{k} := \texttt{k} + \texttt{1}; \} \tag{1.1}$$

For the loop, the assertion $x = 2k$ is a loop invariant but is not inductive. It is not inductive because if we assume that $x = 2k$ holds at the beginning of the loop and calculate the effect of the statements $x := x + y; k := k + 1;$ we cannot derive that $x = 2k$ afterwards, as we do not have enough information about the value of y in the assumption. On the other hand, $x = 2k \wedge y = 2$ is an inductive loop invariant.

A note on notation

Throughout this dissertation, we will use ":=" to denote the imperative state updating assignment, while we will use "=" to denote mathematical equality. The sequencing operator will be ";", and "$*$" will denote non-deterministic choice. Non-deterministic choice is frequently used in program reasoning as a safe approximation to conditional guards that cannot be precisely analyzed, in which case, we assume that both branches can be taken.

It is straightforward to observe that a *given* assertion can be checked/validated to be a correct inductive loop invariant using SMT solving. For instance, we can check whether the candidate assertions $x = 2k$ and $x = 2k \wedge y = 2$ are valid invariants I for the loop. To do that, we simply encode the definition of an inductive loop invariant as formal constraints. One way to formally reason about an assignment $x := e$ is to treat it as an equality between the output value of x, notated as x', and the expression e computed over the inputs. Thus, a set of assignments constitute a *transition* that takes input values to output values of the variables. In our example, there are two paths of sequences of statements that start and end at either an invariant or program entry or exit points. One starts at the beginning of the program (with assertion *true*) and leads up to the loop (with assertion I), and another goes around the loop (starting and ending with assertion I). For these paths, we get the following constraints:

$$\begin{aligned} true \wedge x' = 0 \wedge k' = 0 \wedge y' = 2 &\Rightarrow I' \\ I \wedge k' = k + 1 \wedge x' = x + y &\Rightarrow I' \end{aligned} \tag{1.2}$$

Notice how the consequents are also raised to the output, primed, values. This forwards reasoning approach is similarly used in SSA [5] or symbolic execution [165]. Alternatively, Hoare's rule for assignments [149] can be used for backwards reasoning, and is plausible for the case of verification (Chapter 2). The SSA-style forward approach additionally works for program synthesis where the statements are unknown, and alleviates problems with attempting to substitute *into* unknowns (as in Chapters 3, 4, and 5).

While *checking* that a given assertion is an inductive loop invariant is reducible to SMT queries, as we have seen, it is not obvious how SAT/SMT solving can be used to *infer* loop invariants. Inference using SAT/SMT solving is one of the key technical contributions of this dissertation.

Encoding invariant inference *as SAT/SMT solving* For a given SAT/SMT instance a satisfiability solver computes two values: a binary decision about whether the instance is "sat" or "unsat", and an optional model in the case of satisfiable instances. A model is a value assignment to the unknown variables that leads to the instance evaluating to *true*, e.g., for the case of a SAT instance the model is a assignment of boolean truth values to the propositional variables in the formula. Previous uses of SAT/SMT solvers in invariant validation only use the binary "sat/unsat" decision to check the correctness of the guess for the invariant. More broadly, in program analysis the models from SAT solvers have been used previously to derive counterexamples that explain faults [222, 200, 51, 269, 27, 188, 122].

Our approach is different in that we encode *all valid invariants* as solutions to the satisfiability instance. The model generated by the SAT/SMT solver can then be directly translated to an invariant. The key to doing this is to assume a structural form—i.e., a template, which we discuss in detail later—for the invariant. Then each component in the chosen structure of the invariant is associated with a *indicator* boolean variable. Values, *true* or $false$, for the variables indicate the presence or absence of the component, respectively. Constraints, i.e., clauses in the satisfiability instance, are generated over these boolean indicators from the program being verified. Solving the satisfiability instance gives us the model, i.e., values of the boolean indicators, which are used to reconstruct the actual invariant of the assumed structure. Notice that a model only exists if the instance generated is actually satisfiable. If the instance is unsatisfiable, it implies that no invariant exists of the chosen structural form, i.e., one which is an instantiation of the given template.

A characteristic of a satisfiability-based invariant inference approach is that if there are multiple invariants, the solver finds *one* valid solution that corresponds to one valid invariant. This suffices for program verification, as any inductive invariant proves the required properties, but not for specification inference where we want the best, i.e., weakest or strongest, restrictions on the input or output, respectively. Next, we describe how we can augment the basic approach to generate the weakest/strongest invariants and pre/postconditions for specification inference.

Extending to specification inference Once we have the ability to encode inference as a satisfiability query, it opens the door to *inferring* properties of programs. We can infer preconditions that ensure desired properties of the program's execution, or preclude bad executions. Similarly, we can infer postconditions that hold of program executions. This application highlights a key difference between the mode of use of SMT solvers in this dissertation from that of previous approaches. We can encode pre- and postcondition generation as the inference of an additional invariant at the beginning or end of the program, respectively. The technical developments for invariant inference are correspondingly put to use in deriving specifications, i.e, pre- and postconditions.

Not only that, we can even encode that the desired facts are maximally best, i.e., preconditions are maximally weak and postconditions are maximally strong, which ensures that any other valid pre- or postcondition can be derived from them. This is a non-intuitive application of solvers that have a binary output, and it requires the introduction of other key ideas, namely templates and local encodings, which we describe later (Section 1.1.4).

Automatically deriving pre-/postconditions or specifications is useful as it gives insights into the behavior, good or bad, of the program. For instance, our tool can automatically analyze Binary Search to infer it is only functionally correct if given a sorted input array. It can also analyze Selection Sort to infer that the worst-case number of swaps happen when it is given an array that is almost completely sorted, except that the last element is smaller than the rest. We derive descriptions of behavior that are provably correct (because they are formal and have corresponding invariants associated with them) yet readable (because we infer the least restrictions conditions). Such a tool that is automated, infers expressive properties that are proven formally correct, and outputs readable descriptions has the potentially to significantly help the developer in debugging and interface design.

1.1.2 Satisfiability for Synthesis

Program synthesis is the task of automatically generating a program that matches a given specification. We consider specifications that are mathematical descriptions of the input-output behavior, and also alternative specifications, e.g., as the relationship of a program to another program or as input-output examples.

Program synthesis and program reasoning are in intimately related. If a technique cannot reason about a program specification, given the program, there is no hope of synthesizing a program that meets the specification. Additionally, the provided specification has to be relatively complete so that the synthesizer generates only relevant programs. Such full functional specifications are typically expressed using quantifiers, and therefore we need an expressive reasoning technique, such as the one we develop in this dissertation, to build our synthesizer on top of.

We use two forms of program reasoning techniques, which lead to synthesizers with differing characteristics. Our first technique, *proof-theoretic synthesis*, builds directly off program verification tools and therefore provides formal guarantees about the synthesized program. Our second technique, *path-based inductive synthesis (*`PINS`*)*, leverages symbolic testing—which can be seen as an approximation to formal verification—for synthesizing programs that are correct up to the guarantees that testing provides.

Advantages of a satisfiability-based framework for synthesis As we will see, one of the key requirements of a synthesizer is the need to simultaneously reason about program structure, correctness, and termination. In a satisfiability-based framework, these just correspond to additional clauses in the SAT instances. One can even add clauses corresponding to performance, restrictions on environment interaction (e.g., messages exchanged, information leaked, or locks acquired), resource (e.g., CPU, memory) utilization, and other defining characteristics of the desired program. In this dissertation though, we restrict attention to the core requirements (structure, correctness, and termination). Such combinations are not feasible in traditional approaches to verification and hence we feel that a satisfiability-based reasoning framework is a key facilitator for automatic program synthesis.

1.1.3 Templates

In this section, we elaborate on the key role played by templates in our satisfiability-based approach. Templates restrict attention to a relevant space, be it the space of invariants in reasoning or the space of programs for synthesis. Such restrictions are essential, as the space of *all* possible proofs/programs is likely to remain intractable no matter how sophisticated our theorem proving technology becomes.

Templates provide the form of the desired entities we wish to mechanically infer. For instance, in the case of verification, we intend to infer invariants that provide the proof of correctness of programs. In this case, the technique takes as input a template form (i.e., an expression with holes "$[-]$") for the expected invariants. For example, a template $\vee^2(\wedge^3[-])$ indicates that the invariants contain at most three conjuncts inside each disjunct, of which there can be at most two. A template $\forall(\wedge^3[-] \Rightarrow \wedge^3[-])$ can be used to infer quantified invariants. Similar templates are used to specify the desired form of inferred preconditions and postconditions. In the case of synthesis, scaffolds are templates for desired programs.

Note that templates do not describe specific structures (invariants or programs), but rather their class. In this regard, they are analogous to *abstract domains*, which have been used in earlier approaches to program reasoning, e.g. in abstract interpretation [72, 33, 74], and model checking [98]. Templates can be viewed as an optimized approach to lifting domains to more expressive relations. For instance, while a template $\forall(\wedge^3[-] \Rightarrow \wedge^3[-])$ can be very efficiently handled in our system because of its restricted structure (that the user guessed), it can be viewed as a specialization of the domain for the holes, lifted to disjunction, and additionally quantification. Such a general domain will be very inefficient, if at all the theoretical machinery can be built, and consequently not practical. Additionally, we find that the expressivity afforded by templates facilitates not only reasoning but also program synthesis.

For example, a widely used domain is the octagon domain [202], which can specify facts between two variables, x and y, of the form $\wedge_i(\pm x \pm y \leq c)$. On the other hand, templates allow us to specify not just conjunctions, but also atomic facts such as $c_0+c_1x+c_2y+c_3z\,..\geq 0$, wrapped inside *arbitrary* boolean connectives, e.g., disjunctions and even quantifiers. The difference in the expressivity of an octagon domain and a linear arithmetic template is illustrated in Figure 1.1.

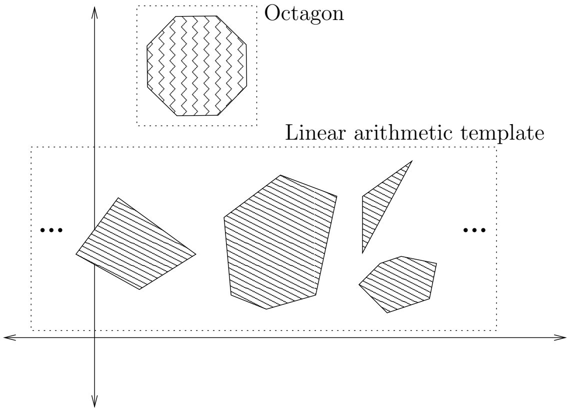


Figure 1.1: The expressivity of the octagon domain vs. linear arithmetic templates.

While strictly more expressive for *given* facts, in general templates are incomparable to domains because for templates the outerlevel form is more strictly specified. For instance, an octagon domain can represent any arbitrary finite number of conjuncts of a limited form, while a template requires a finite upper bound on the number of conjuncts (in each disjunct, and an upper bound on the number of total disjuncts in the DNF representation). In practice though, the finite bounds can be chosen to be large enough to capture all expected facts.

The use of templates parameterized with finite bounds introduces tradeoffs between expressivity and efficiency, which the user can tune. While a template with larger bounds allows for more expressive invariants, the corresponding satisfiability instances that need to be solved are proportionally bigger, which in some cases also means that they are harder to solve. With the current state-of-art it is most prudent to have the user guess the template parameters. In the future we expect it will be feasible iteratively explore the space of the parameters automatically.

1.1.4 Maximally Best Solutions using Satisfiability

The use of templates enables us to compute optimal, i.e., maximally best, values required for certain problems in a satisfiability-based framework. The key insight is based on *local reasoning* (in the proof) and *finite satisfiability encoding* (of local constraints). For instance, we can compute maximally weak preconditions and maximally strong postconditions using a finite encoding into satisfiability.

This is a novel application of satisfiability solvers whose search for a satisfying solutions does not have any particular monotonicity property and may output *any* satisfying solution. To get optimal solutions we need to ensure through appropriate constraints that *every* satisfying solution is a local maxima.

Example 1.2 *Figure 1.2 shows a lattice whose elements are two linear inequalities, over variables x and y, conjuncted together. We concentrate on the lattice point $x - y \geq 0 \wedge x + y \leq t$. If we assume a template that can only represent constants of a certain maximum size, say c, then it tells us what the smallest possible deviation (shifting or rotation) can be. In particular it will be related to the smallest possible constant, i.e., $1/c$, expressible under this assumption. We can finitely enumerate the*

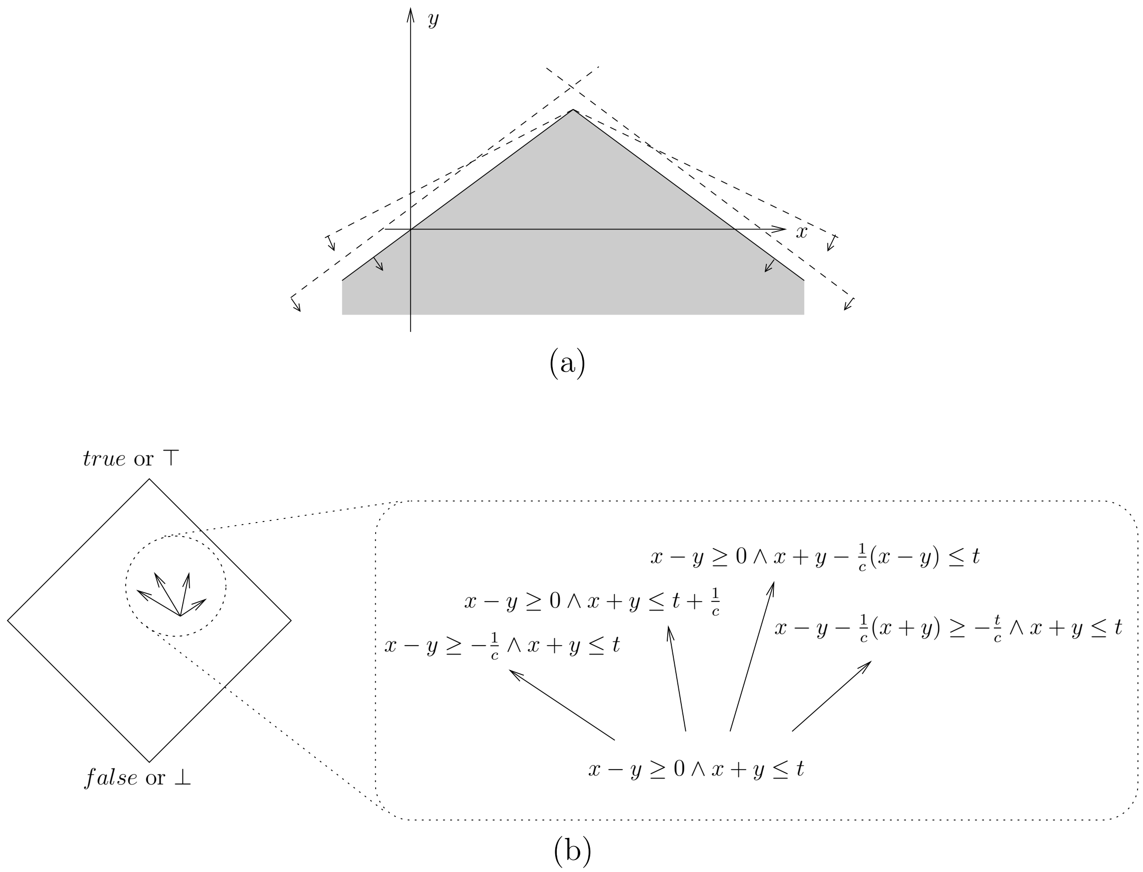

y
x
(a)
true or $\top$
false or $\bot$
$x - y \geq 0 \wedge x + y - \frac{1}{c}(x - y) \leq t$
$x - y \geq 0 \wedge x + y \leq t + \frac{1}{c}$
$x - y - \frac{1}{c}(x + y) \geq -\frac{t}{c} \wedge x + y \leq t$
$x - y \geq -\frac{1}{c} \wedge x + y \leq t$
$x - y \geq 0 \wedge x + y \leq t$
(b)

Figure 1.2: Templates facilitate enumerating local neighbors (dashed lines). Shown here is the case of facts of the form $a \wedge b$, where a and b are linear inequalities over variables x and y. (a) Each fact (lattice point) induces an area in the x, y-graph. The shaded area denotes the lattice point $x - y \geq 0 \wedge x + y \leq t$, with each inequality being the bold line at the boundary of the area. Four lattice points immediately weaker than this fact exist, as shown by the dotted lines. We get two local (weaker) neighbors by *shifting* one of the inequalities by a small amount. We get another two local (weaker) neighbors by *rotating* one of the inequalities by a small angle. (b) Partial order lattice, with elements that are conjunctions of linear inequalities, ordered by the implication relation. We expand out the original fact, and its four immediately weaker neighbors in the lattice that can be enumerated because of restrictions on the maximum constant c representable in the system.

local neighbors (only because of the presence of a template) and therefore construct a finite encoding of maximal/minimal optimality. For instance, in the example above, we can constrain the system to say that each immediately weaker neighbor is not "valid" while the current lattice point is "valid"—whatever the notion of validity is. Then just by solving the satisfiability instance, we will generate points in the lattice that are maximal, i.e., they are valid while any immediate weaker points are not. This would not have been possible without a template, as we would not know what the least weakening is.

1.2 Program reasoning with linear arithmetic

While linear arithmetic expressions are relatively simple to reason about mechanically, they have many applications in program reasoning and—as we see in this dissertation—in program synthesis. Linear arithmetic can be used to reason about a wide variety of program properties through suitable modeling. For instance, not only does it suffice for a fair majority of interesting invariants required for proving memory safety or termination, but we can also reason about the size of data structures with insert/delete operations, or array bounds checks using linear arithmetic.

Templates over linear arithmetic are atomic linear relations wrapped within an arbitrary boolean structure. In this dissertation, for linear arithmetic we consider only the boolean structures without quantification, i.e., limited to conjunctions and disjunctions. Quantification is handled for predicate abstraction, described later. Negation is at the innermost level, and is encoded by suitably modifying the atomic inequality. Without loss of generality, we assume that the atomic linear relations are of the form $c_0 + c_1 x + c_2 y + c_3 z \ldots \geq 0$, where x, y, z are program variables, c_i's are integer coefficients, and the boolean structure is described using disjunctive normal form (DNF).

Our linear arithmetic templates are parameterized by two integer values: the maximum number of disjuncts in the outermost disjunction, and the maximum number of conjuncts in each disjunct. For example, a template with 2 disjuncts and 3 conjuncts can model the formula $(x = y \wedge x > 10) \vee (x > y \wedge y \leq z \wedge x \leq 0)$.

Example 1.3 *For inferring an invariant for the program in Example 1.1, a plausible template could be conjunctions, let us say five in number, of linear inequalities between the variables, $c_0 + c_1 x + c_2 y + c_3 k \geq 0$. Here c_i's are the unknown (integer) variables. Notice that the invariant $x = 2k \wedge y = 2$ can be embedded in this template as $(x - 2k \geq 0) \wedge (-x + 2k \geq 0) \wedge (y - 2 \geq 0) \wedge (-y + 2 \geq 0) \wedge (1 \geq 0)$. The last term is one way to encode true as a linear relation. Using Eq. (1.2), we generate (integer) constraints over the c_i's. We then assume a bit-vector representation of a suitably large size for each of the integer unknowns, and generate a SAT instance over boolean indicator variables. We can directly read off the invariant, from the solution to this instance.*

Notice that a template with at least four inequalities can express the inductive invariant $x = 2k \wedge y = 2$. On the other hand, if the template had fewer inequalities then the SAT instance generated would be unsatisfiable. Two inequalities can encode $x = 2k$, but this fact is not inductive and therefore the boolean clauses generated for *the second constraint in Eq. (1.2) will make the SAT instance unsatisfiable.*

1.3 Program reasoning with predicate abstraction

While linear arithmetic is good for certain classes of properties, in some cases reasoning and synthesis is best described by more expressive predicates. For instance, consider reasoning about the contents of arrays or linked data structures (lists, trees). A standard approach to modeling array reads and writes is through McCarthy's *select/update predicates*. Linked data structures can be modeled using *reachability predicates*. Such functional modeling of programming constructs using predicates lends itself well to mechanization through SMT solvers. Predicate abstraction [128] is an approach that can reason using arbitrary predicates, as long as the underlying theorem prover/SMT solver knows how to interpret the operators used. In this dissertation, we show how to do satisfiability-based reasoning and synthesis over predicate abstraction.

Let us elaborate more on the use of predicates for encoding the semantics of programming constructs. For arrays, the standard approach uses McCarthy's `sel`/`upd` predicates. For array A, location i, and value v, the predicate $\texttt{sel}(A, i)$ returns the contents at i, and $\texttt{upd}(A, i, v)$ returns a *new* array with the contents at i updated to be v. These predicates are related by the axiom:

$$\forall A, i, j, v : \texttt{sel}(\texttt{upd}(A, i, v), j) \quad = \quad \texttt{if}\ (i = j)\ \texttt{then}\ v\ \texttt{else}\ \texttt{sel}(A, i)$$

A version of this axiom originally appeared in McCarthy's paper [198], and solvers implement decision procedures that efficiently check the validity of formulae under this axiom, e.g., Z3's implementation uses additional combinators [87].

Templates over predicate abstraction consist of a boolean structure (potentially with quantifiers) that contain holes. Each hole is populated with a subset, representing conjunction, of predicates from a given universe Π_p. Recall that predicate abstraction represents the abstract states of a program as subsets of the predicates that hold in the state, and the predicates come from $\Pi_p = \{q_1, q_2, .., q_n\}$. Each q_i can be arbitrarily expressive as long as the underlying theorem prover/SMT solver understands the operators used. An example of a predicate set is $\Pi_p = \{(x = \texttt{sel}(A, i)), (i \geq z + 1), (A' = \texttt{upd}(A, z, v))\}$.

In this dissertation, templates over predicate abstraction are specified as the outer boolean structure. For example, $\forall([-] \Rightarrow [-])$ is a generic, fairly expressive, template that we use frequently. Each of the holes $[-]$ are populated by the system with appropriate conjunctions of predicates from subsets of Π_p.

Not only does our approach to program reasoning leverage the engineering advances made in SAT/SMT solving, through the use of appropriate templates it also allows us to infer expressive invariants that were beyond the reach of previous approaches. For instance, we can use this approach to infer *quantified invariants* that are facts with universal or existential quantification. Quantified invariants are very useful in expressing properties of programs manipulating unbounded data structures where we need to quantify over all elements. Examples of such data structures that could be of unbounded sizes are arrays, lists, and trees.

Example 1.4 *Using predicate abstraction, we can prove selection sort correct by inferring the following invariant:*

$$
\begin{array}{l}
i < j \wedge i < n-1 \quad \wedge \\
\forall k : i \leq k < n-1 \Rightarrow A[n-1] < A[k] \quad \wedge \\
\forall k : i \leq k < j \Rightarrow A[min] \leq A[k] \quad \wedge \\
\forall k, k' : i \leq k < k' < n-1 \Rightarrow A[k] \leq A[k']
\end{array}
$$

where n is the size of the array A being sorted, i and j are the loop counters for the nested loops, and min is the index location of relevant minimum element. The templates used are $[-]$, $\forall k : [-] \Rightarrow [-]$, and $\forall k, k' : [-] \Rightarrow [-]$, and the predicates are $\alpha < \beta$ and $\mathtt{sel}(A, \alpha) < \mathtt{sel}(A, \beta)$ (and $\leq$'s), where α and β are instantiated with programs variables (i, j, n and min), quantifier bound variables (k and k'), and their offsets (± 1).

1.4 Verification-inspired synthesis

Given a formal specification and constraints on the structure of the desired program, *proof-theoretic synthesis*, inspired by verification, simultaneously generates not only a program but also the corresponding proof of correctness. The proof is a witness to the fact that the program meets its specification.

The key observation that enables building synthesizers out of verifiers is that when reasoning using the transition system representation, statements are just *equality predicates*. So if our verifier can reason using this representation with known equalities (for the statements), we can potentially use it to *infer* not only the invariant facts, but also the equality facts corresponding to the statements!

Example 1.5 *Let us revisit the program from Example 1.1 and constrain its output. Specifically, let us say that we expect the program to terminate in a state in which* $x = 2n$*, where* n *is some input to the program. If the loop guard is* $x \leq n$ *instead of non-deterministic choice then the program does indeed compute* $x = 2n$*. Written using a transition system representation, the program constraints are:*

$$\begin{aligned} \mathit{true} \wedge S_1 &\Rightarrow I \\ I \wedge G \wedge S_2 &\Rightarrow I \\ I \wedge \neg G &\Rightarrow x = 2n \end{aligned} \tag{1.3}$$

where

$$\begin{aligned} S_1 &\doteq x = 0 \wedge k = 0 \wedge y = 2 \\ S_2 &\doteq x' = x + y \wedge k' = k + 1 \\ G &\doteq x \leq n \end{aligned} \tag{1.4}$$

During invariant inference (for reasoning about the given program), each of these constraints had unknown I *and known* S_1, S_2*, and* G *as shown. In this representation,* S_1, S_2*, and* G *are logical facts that can potentially also be inferred by the program reasoning tool, along with* I*, i.e., the proof. So our hope is to send the constraints Eq. (1.3) to existing solvers and get a solution for* S_1, S_2, G *and* I*, such as Eq. (1.4).*

Our optimism may be premature because not all solutions to this underconstrained system of constraints will be valid programs. The first concern is that the semantics of statement and guard unknowns are not enforced. Notice that an assignment of $S_1 = S_2 = I \doteq false$ is a valid solution, but it does not correspond to any valid assignments. Transitions S_1 and S_2, which are conjunctions of equalities between outputs and expressions over inputs, can never be $false$, and therefore this solution cannot be translated to any assignments of the form $x := e$. Correspondingly, constraints are needed for guard unknowns such that solutions translate to valid control flow. We call these constraints, which ensure that solutions correspond to valid imperative programs the *well-formedness constraints*.

Solving safety constraints (Eq. (1.3)) together with well-formedness constraints indeed yields valid imperative programs, but it does not preclude trivial solutions. For instance, a solution $G = I \doteq true$ satisfies the constraints. In fact, $G \doteq true$ corresponds to a non-terminating loop. We need to eliminate such uninteresting programs, and we therefore also assert *termination constraints*.

Solving safety, termination, and well-formedness constraints, together called *synthesis conditions*, yield valid imperative terminating programs that meet the specification and have a corresponding correctness proof. In this dissertation, we show that these constraints can be written in a form amenable to solving by current verifiers, thereby bringing engineering advances in verification to synthesis. We show that satisfiability-based verifiers can be used *unmodified* as program synthesizers.

The input to our synthesizer is a *scaffold* of the desired computation. The scaffold provides the looping structure (e.g., does the program contain a nested loop, or two loops in a sequence), constraints on resources (e.g., number of variables), and domain of core operations (e.g., operators, or function calls available). For example, we may wish to preclude multiplication as one of the operators in the previous program, because otherwise the synthesizer may generate $x := 2n$ and terminate immediately. With only linear operators, the synthesizer will be forced to generate a loop. From the scaffold we generate synthesis conditions, which we solve using satisfiability-based program verifiers. We have been able to synthesize programs such as Strassen's matrix multiplication, Bresenham's line drawing algorithm, dynamic programming examples, and all major sorting algorithms just from their scaffold specifications.

Proof-theoretic synthesis leverages the connections between automatic program verification and automatic program synthesis. If we have a verifier that can reason about programs over a particular domain, then it can be used as a corresponding synthesizer for that domain, taking as input a scaffold, and solving additional constraints described by synthesis conditions.

1.5 Testing-inspired synthesis

Given a functional specification, *path-based inductive synthesis* (`PINS`), inspired by testing, leverages symbolic testing to synthesize programs. `PINS` is a more pragmatic synthesis approach since it does away with (potentially complicated and expensive) formal invariants, and instead uses program paths to reason about behavior and to synthesize. Additionally, the functional behavior of certain programs can be specified as their relation to another program, which alleviates the need for formal descriptions of the functional specification.

The key observation that enables building synthesizers out of testing tools is that if a program is functionally correct on a set of paths through it, then it is either correct or at least “close to” correct for all paths. We apply this intuition to program synthesis by ensuring that the synthesized program meets the specification on some set of paths. By increasing the number of paths, we are able to eliminate invalid programs, until only one valid solution remains that is correct for all paths explored. Additionally, we impose stronger constraints on the program statements by testing the paths symbolically, as opposed to with concrete values. For instance, for a program that takes x as input, instead of constraining the behavior on $x = 1, x = 2, x = 3, x = 4$, and so on, we instead run the program with a symbolic value α for x, with the side condition that $\alpha > 0$. Thus, a path explored with symbolic inputs captures the behavior of the program over multiple concrete inputs that take the same path.

Let us first describe the input to the `PINS` algorithm. Suppose first that we have a structure for the unknown program and its expected specification. A structure for an unknown program is a description of its control flow, with unknown conditional and loop guards and statements. For the program in Example 1.5, $S_1; \texttt{while}(G)\{S_2\}$ is a potential structure with S_1, S_2, and G as unknowns, and its expected specification is $x = 2n$. Not everything is required to be unknown. Another potential template is $S_1; \texttt{while}(x \leq n)\{S_2\}$. That is, a template is a *partial program* in which the synthesizer fills in the unknown holes.

We now describe the core technique behind `PINS`. For a given partial program, we can choose certain paths through it and constrain that the specification is met on each of those paths.

Example 1.6 *For the partial program $S_1;$`while`$(G)\{S_2\}$ we can write down constraints for three paths, one that does not enter the loop, and two that go through the loop once and twice, as follows:*

$$\begin{aligned} true \wedge S_1 \wedge \neg G' &\Rightarrow x' = 2n' \\ true \wedge S_1 \wedge G' \wedge S_2' \wedge \neg G'' &\Rightarrow x'' = 2n'' \\ true \wedge S_1 \wedge G' \wedge S_2' \wedge G'' \wedge S_2'' \wedge \neg G''' &\Rightarrow x''' = 2n''' \end{aligned} \tag{1.5}$$

Notice that every time control passes through a statement block, S_1 or S_2, every subsequent read uses more primes—in line with the transition system semantics. Notice that if we had used concrete execution, each one of these constraints would have been expanded to multiple constraints for particular values of the input variables that follow those paths.

The advantage of using paths to generate safety constraints is that the system need not reason about invariants, which can potentially be very complicated. The disadvantage is that, in the presence of loops, the number of paths is unbounded. So the approach can only be complete up to a certain confidence level, which rises with the number of paths. With these constraints as proxies for invariant-based safety constraints, we can use the technology already developed to solve for the unknowns and synthesize programs.

As with any testing-based approach, we need to worry about which paths to explore. Notice that there could be multiple programs that satisfy the constraints for a limited set of paths. For example, the first constraint in Eq. (1.5) imposes no restrictions on S_2, and therefore if we were to only consider that constraint then all values for S_2 are valid. So we need to explore more paths to eliminate invalid programs. A naive approach would be to explore random paths, but this fails as expected, due to combinatorial explosion. The situation is exacerbated in the presence of unknown guards and statements.

We devise a *directed path exploration* scheme that infers relevant paths. A path is relevant if it eliminates specific invalid programs from the space of solutions. The path exploration scheme picks one solution program (which satisfies the safety constraints on paths explored until that point) and instantiates the partial program with that solution. It then finds a path in the partial program such that it is feasible for the instantiated program. If the chosen solution does not correspond to a program that meets the specification, adding this new path to the system eliminates the solution with high probability. This is the case because the path is feasible with respect to the solution and therefore it is unlikely that the instantiated program will meet the specification if it is invalid. On the other hand, if the chosen solution corresponds to a program that meets the specification, then adding this new path will only reinforce the solution. Thus by iteratively selecting a solution from the space remaining and using directed path exploration to prune out invalid programs, we eventually narrow the space down to only the valid programs.

`PINS` is a general synthesis technique that works without referring to formal invariants, but does need a formal specification. We consider its application to cases where the specification is trivial or mechanically derivable. Consider the case of program inversion. In program inversion, the sequential composition of a known program with its (unknown) inverse has the trivial identity specification. Also, typically the structure of the inverse, but not the exact computations, is similar to the given program. Therefore, we mine the template for the inverse and apply `PINS` to automatically synthesize the precise operations of the inverse. Additionally, we also consider parallel composition and apply `PINS` to automatically generate clients from servers.

Path-based inductive synthesis (`PINS`) shows how testing can be viable approach to program synthesis. Intuitively, it exploits the pigeonhole principle by exploring more paths than can be individually explained by the template, i.e., partial program. While the core approach shows that synthesis is feasible using testing random paths, for it to be efficient in practice, a direct approach to path exploration is required.

1.6 Engineering Verifiers and Synthesizers

In the previous sections we have gave an overview of the theoretical insights that go into using a satisfiability-based approach (along with templates) to do expressive program verification and even to synthesize programs. We have built the VS^3—<u>V</u>erification and <u>S</u>ynthesis using <u>S</u>MT <u>s</u>olvers—suite of tools that implement these ideas. While the core satisfiability-based approach is itself novel, due to the non-traditional analysis mechanism employed this approach opens up avenues for engineering optimization that were previously not present. We have been able to build tools that meet, if not consistently outperform, previous tools in terms of efficiency, while being able to handle much more expressive reasoning. We have demonstrated the proof-of-concept by employing the VS^3 tools to verify standard difficult benchmarks in verification; and for the first time automatically synthesize programs from high level specifications.

1.7 Key Contributions and Organization

This dissertation makes the following contributions:

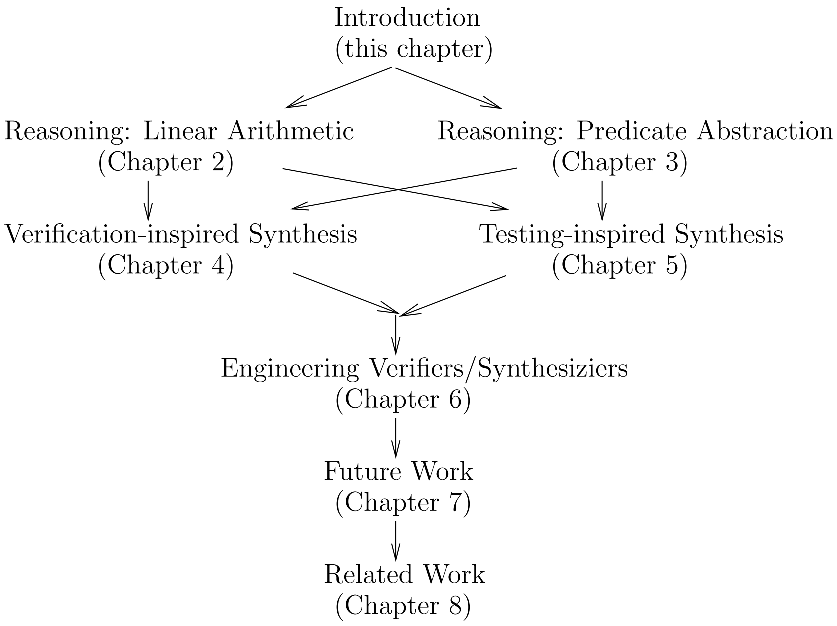
- We present an approach for encoding proofs for program correctness, i.e., invariants, as (arbitrary) solutions to propositional satisfiability instances. This facilitates finding these proofs using off-the-shelf SAT solvers. We also present extensions that allow us to encode specification inference in the same framework.

- We show how program synthesis can be viewed as generalized program verification, thereby allowing the use of certain automatic verifiers as automatic synthesizers. Thus, if we have a verifier with specific properties, that can prove programs correct in a particular domain, by this approach we have a corresponding synthesizer that can automatically generate programs in that domain as well.

- We extend the idea of template-based analyses to expressive program reasoning and program synthesis. Templates have two benefits. One, they make the task of the automatic tool tractable by limiting the search for proofs, specifications, and programs to particular forms. Note that these forms are not specific to the programs being reasoned about or synthesized, but concern a category of programs. Two, they serve as a specification mechanism by which the user can limit the types of proofs, specifications, or programs desired.

- We show that in the context of a template-based approach, we can synthesize programs without formal specifications or proofs. As testing can be viewed as a means of approximate verification, in a similar vein this approach can be viewed as a means of approximate synthesis.

Chapter dependencies While the developments in this dissertation follow an almost linear progression, each chapter starts with an overview of the key results presented therein. The dependencies across chapters are as follows:


Introduction
(this chapter)
Reasoning: Linear Arithmetic
(Chapter 2)
Reasoning: Predicate Abstraction
(Chapter 3)
Verification-inspired Synthesis
(Chapter 4)
Testing-inspired Synthesis
(Chapter 5)
Engineering Verifiers/Synthesiziers
(Chapter 6)
Future Work
(Chapter 7)
Related Work
(Chapter 8)

Chapter 2

Program Reasoning over Linear Arithmetic

*"Our path is not going to be linear
or smooth. It's still early days."*
— Mark Fields[1]

In this chapter we present a satisfiability-based approach for modeling a wide spectrum of program analyses in an expressive domain containing disjunctions and conjunctions of linear inequalities. In particular, we show how to model the problem of context-sensitive interprocedural program verification. We also present the first satisfiability-based approach to maximally weak precondition and maximally strong postcondition inference. The constraints we generate are boolean combinations of quadratic inequalities over integer variables. We reduce these constraints to SAT formulae using bit-vector modeling and use off-the-shelf SAT solvers to solve them.

Furthermore, we present interesting applications of the above analyses, namely bounds analysis and generation of most-general counterexamples for both safety and termination properties. We also present encouraging preliminary experimental results demonstrating the feasibility of our technique on a variety of challenging examples.

[1] American Footballer, 1972–.

2.1 Using SAT Solvers for Invariant Inference

Program reasoning consists of verifying the correctness of programs or inferring pre- and postconditions (which are semantic descriptions of program properties). The key difficulty in program verification is the task of inferring appropriate program invariants, i.e., facts that hold at program points whenever control reaches those points. Inferring program properties can be seen as an extension of verification, where in addition to the invariants, the pre- or postconditions are also inferred.

Discovering inductive program invariants is critical for both proving program correctness and finding bugs. Traditionally, iterative fixed-point computation based techniques like data-flow analyses [164], abstract interpretation [72] or model checking [98] have been used for discovering these invariants. An alternative is to use a *constraint-based invariant generation* [62, 75, 42, 219] approach that translates the second-order constraints that a program induces into first-order quantifier-free constraints that can be solved using off-the-shelf solvers. While previous constraint-based approaches employed mathematical solvers for finding solutions to the resulting constraints [62, 75], in this chapter we propose using SAT solvers, i.e., a *satisfiability-based invariant generation approach*. The last decade has witnessed a revolution in SAT/SMT based methods enabling solving of industrial sized satisfiability instances. This presents a real opportunity to leverage these advances for solving hard program analysis problems.

Constraint/satisfiability-based techniques offer two other advantages over fixed-point computation based techniques. First, they are goal-directed and hence have the potential to be more efficient. Second, they do not require the use of widening heuristics that are used by fixed-point based techniques and lead to loss of precision that is often hard to control.

Here, we describe satisfiability-based techniques over linear arithmetic for three classical program analysis problems, namely *program verification*, *maximally weak precondition* generation and *maximally strong postcondition* generation. Using this core framework of analyses we further show interesting applications to bounds analysis and finding most-general counterexamples to safety and termination properties. The key contributions are in the uniform satisfiability-based approach to core program analyses (Sections 2.2–2.5) and their novel applications (Section 2.7). We have also implemented these ideas in a tool that we call $\texttt{VS}^3_{\texttt{LIA}}$. A distinguishing feature of $\texttt{VS}^3_{\texttt{LIA}}$ is that it can uniformly handle a large variety of challenging examples that otherwise require many different specialized techniques for analysis.

The goal of *program verification* is to discover invariants that are strong enough to verify given assertions in a program. We present a satisfiability-based technique that can generate linear arithmetic invariants with arbitrary *boolean structure* (Section 2.2), which also allows us to extend our approach to a *context-sensitive interprocedural setting* (Section 2.3). A key idea of our approach is a scheme for reducing second-order constraints to SAT constraints; this can be regarded as an independent contribution to solving a special class of second order formulas. Another key idea concerns an appropriate choice of cut-set which, surprisingly, has until now been overlooked. $\texttt{VS}^3_{\texttt{LIA}}$ can verify safety properties, provided as assertions, in benchmark programs that require disjunctive invariants and sophisticated procedure summaries.

These programs have appeared as benchmarks for alternative state-of-the-art techniques. We also show how satisfiability-based invariant generation can be applied to verifying termination properties as well as the harder problem of *bounds analysis* (Section 2.7.1).

The goal of *strongest postcondition generation* is to infer the most descriptive/precise postcondition that characterizes the set of reachable states of the program. Current constraint-based invariant generation techniques work well only in a program verification setting, where the problem enforces the constraint that the invariant should be strong enough to verify the assertions. But in absence of assertions in programs, there is no guarantee of the precision of invariants. We describe a satisfiability-based technique that can be used to discover strongest, or more precisely maximally strong, invariants (Section 2.5). Some previous techniques generate precise invariants using widening heuristics that are tailored to specific classes of programs [264, 132, 125, 126]. $\mathtt{VS}^3_{\mathtt{LIA}}$ can uniformly discover precise invariants for all such programs.

The goal of *weakest precondition generation* is to infer the least restrictive precondition that ensures validity of all assertions in the given program. We present a satisfiability-based technique for discovering weakest, or more precisely maximally weak, preconditions (Section 2.4). $\mathtt{VS}^3_{\mathtt{LIA}}$ can generate maximally weak preconditions of safety as well as termination properties for difficult benchmark programs. We do not know of any previous tool that can infer these properties for the programs we consider.

We also describe an interesting application of maximally weak precondition generation: generating *most-general counterexamples* for both safety (Section 2.7.2) and termination (Section 2.7.3) properties. The appeal of generating most-general counterexamples (as opposed to generating any counterexample) lies in characterizing all counterexamples in a succinct specification that provides better intuition to the programmer. For example, if a program has a bug when $n > 200 \wedge 9 > y > 0$, then this information is more useful than simply generating any particular counterexample, say $n = 356 \wedge y = 7$ (Figure 2.11). We have also successfully applied $\texttt{VS}^3_{\texttt{LIA}}$ to generate counterexamples to termination of programs (taken from recent work [142]).

2.2 Program Verification

Given a program with some assertions, the program verification problem is to verify whether the assertions are valid. The challenge in program verification is to discover the appropriate invariants at different program points, especially inductive loop invariants, that can be used to prove the validity of the given assertions. (The issue of discovering counterexamples, in case the assertions are not valid, is addressed in Section 2.7.2).

Program model In this chapter, we consider programs that have linear assignments, i.e., assignments $x := e$ where e is a linear expression, or non-deterministic assignments $x :=?$. We also allow for assume and assert statements of the form $\texttt{assume}(p)$ and $\texttt{assert}(p)$, where p is some boolean combination of linear inequalities $e \geq 0$.

Here x denotes some program variable that takes integral values, and e denotes some linear arithmetic expression. Since we allow for `assume` statements, without loss of generality, we assume that all conditionals in the program are non-deterministic.

2.2.1 Verification Conditions: Program semantics as constraints

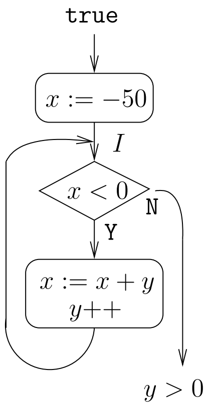
In this section, we describe encoding the semantics of programs as logical constraints. The problem of program verification can be reduced to the problem of finding solutions to a second-order constraint. The second-order unknowns in this constraint are the unknown program invariants that are inductive and strong enough to prove the desired assertions. In this section we describe the conversion of programs to constraints.

Consider the program in Figure 2.1(a) with its control flow graph in Figure 2.1(b). The program precondition is `true` and postcondition is $y > 0$. To prove the postcondition, at some point in the loop such as the one shown, we need to find an invariant I. There are three paths in this system that constrain I. The first is the *entry case* meaning the path from `true` to I. The second is the *inductive case* meaning the path that starts and ends at I and goes around the loop. The third is the *exit case* meaning the path from I to $y > 0$. Figure 2.1(c) shows the corresponding constraints. We now show how to construct these constraints formally.

The first step is to choose a cut-set. A *cut-set* is a set of program locations (called *cut-points*) such that each cycle in the control flow graph passes through some

```
PV2 () {
  x := −50;
  while (x < 0) {
     x := x + y;
     y++;
  }
  assert(y > 0)
}
```

(a)



(b)

$$\begin{aligned}
&\forall_{x,y}\phi(I):\\
&\qquad \texttt{true} \Rightarrow I[-50/x]\\
&\quad I \wedge x < 0 \Rightarrow I[(y{+}1)/y, (x{+}y)/x]\\
&\quad I \wedge x \geq 0 \Rightarrow y > 0
\end{aligned}$$

(c)

Figure 2.1: Illustrating program reasoning over linear arithmetic using an example. (a) Simple example with loop invariant (at the header node) I (b) the control flow graph and (c) the corresponding constraint. The satisfying solution $(x < 0 \vee y > 0)$ to the constraint is disjunctive.

program location in the cut-set. One simple way to choose a cut-set is to include all targets of back-edges in any depth first traversal of the control-flow graph. (In case of structured programs, where all loops are natural loops, this corresponds to choosing the header node of each loop.) However, as we will discuss in Section 2.2.4, some other choices of cut-set might be more desirable from an efficiency/precision viewpoint. For notational convenience, we assume that the cut-set always includes the program entry location $\pi_{\texttt{entry}}$ and exit location $\pi_{\texttt{exit}}$.

We then associate each cut-point π with a *relation* I_π over program variables that are live at π. The relations $I_{\pi_{\text{entry}}}$ and $I_{\pi_{\text{exit}}}$ at program's entry and exit locations, respectively, are set to `true`, while the relations at all other cut-points are unknown relations that we seek to discover. Two cut-points are *adjacent* if there is a *path* in the control flow graph from one to the other that does not pass through any other cut-point. We establish constraints between the relations at adjacent cut-points π_1 and π_2 as follows. Let $\texttt{Paths}(\pi_1, \pi_2)$ denote the set of paths between π_1 and π_2 that do not pass through any other cut-point. We use the notation $\texttt{VC}(\pi_1, \pi_2)$ to denote the constraint that the relations I_{π_1} and I_{π_2} at adjacent cut-points π_1 and π_2 respectively are consistent with respect to each other:

$$\texttt{VC}(\pi_1, \pi_2) \quad = \quad \forall X \left(\bigwedge_{p \in \texttt{Paths}(\pi_1, \pi_2)} (I_{\pi_1} \Rightarrow \omega(p, I_{\pi_2})) \right)$$

Above, X denotes the set of program and fresh variables that occur in I_{π_1} and $\omega(p, I_{\pi_2})$. The notation $\omega(p, I)$ denotes the weakest liberal precondition [92, 130] of path p (which is a sequence of program instructions) with respect to I:

$$\begin{array}{cc}
\omega(\texttt{skip}, I) = I & \omega(\texttt{assume}\ p, I) = p \Rightarrow I \\
\omega(x := e, I) = I[e/x] & \omega(\texttt{assert}\ p, I) = p \wedge I \\
\omega(x :=?, I) = I[r/x] & \omega(S_1; S_2, I) = \omega(S_1, \omega(S_2, I))
\end{array}$$

where r is a fresh variable and the notation $[e/x]$ denotes substitution of x by e. Until the step where the invariant is instantiated as a template, the substitutions need to be accumulated and deferred.

Alternatives to substitution

Here, we present substitution as the means of backwards reasoning, i.e., applying Hoare's axiom for assignment [149]. It is instructive to note that substitution is not a logical primitive, and consequently, invariant inference using theorem provers (that work over a specific logic) can potentially be complicated by the presence substitution. Fortunately, by assuming a *template* for the invariants, substitution into them is feasible.

Using substitution is not critical to the developments in this dissertation. For the rest of the chapter, we will be agnostic to the mechanism used for reasoning about assignment, either backwards using Hoare's assignment rule and templates to substitute into, or forwards using equality predicates (with variable versions, like in single static assignment (SSA)—developed for compiler optimizations by Wegman, Zadeck, Alpern, Rosen [4, 230]—or symbolic execution [165]). In fact, in all subsequent chapters, we will use equality predicates because of two reasons. First, it alleviate the inconvenience of substituting into unknowns, and second, for the case of synthesis the variable being assigned to is also unknown. We describe this approach in more detail in Chapter 3, Section 3.3.3.

Let π_1, π_2 range over pairs of adjacent cut-points. Then any solution to the unknown relations I_π in the following verification constraint (which may also have substitutions) yields a valid proof of correctness.

$$\bigwedge_{\pi_1,\pi_2} \mathtt{VC}(\pi_1, \pi_2) \tag{2.1}$$

This constraint is implicitly universally quantified over the program variables and is a function of $\vec{I}$ (the vector of relations I_π at all cut-points including $I_{\pi_{entry}}, I_{\pi_{exit}}$). We therefore write it as the *verification condition* $\forall X.\phi(\vec{I})$. For program verification $I_{\pi_{entry}}$ and $I_{\pi_{exit}}$ are set to `true`. Going back to the example, the second-order constraints corresponding to the program in Figure 2.1(a) are shown in Figure 2.1(c) and correspond to the entry, inductive, and exit constraints for the loop.

2.2.2 Template specification T

We define the notion of a template specification T over linear arithmetic inequalities inside arbitrary boolean conjunctions and disjunctions. For the sake of simplicity and without loss of generality, we assume that the template is expressed in disjunctive normal form (DNF) and negations are at the innermost level, and can therefore be encoded in the linear term by appropriate manipulations. In later chapters, we will use a more expressive template, e.g., containing quantifiers (Chapter 3), and even templates for control flow of programs (Chapter 4).

For the purposes of this chapter, a template specification consists of two elements. The first is the boolean DNF structure, indicated by $\mathit{template}(T)$, and is just a pair of integers (d, c) that indicate there are d disjuncts in the formula with c conjuncts each. The second is the maximum size of constants represented in binary format, indicated by $\mathit{bvsize}(T)$.

Example 2.1 *Consider the template specification T with template$(T) = (3, 2)$ and bvsize$(T) = 11$ for a program with program variables x, y. In this case the general form of invariants for this template specification is:*

$$\bigvee_{j=1..3} (c^j_0 + c^j_1 x + c^j_3 y \geq 0) \wedge (c^j_4 + c^j_5 x + c^j_6 y \geq 0)$$

where the $c^j_{0..6}$'s are the constants in the j^{th} disjunct and can represent integers between -1024 and 1023 with a two's complement representation using 11 bits.

2.2.3 Constraint solving

In this section we show how to solve the second-order constraint from Eq. 2.1 that represents the verification condition of unknown relations at cut-points. The key idea is to reduce the second-order constraint into a boolean formula such that a satisfying assignment to the formula maps to a satisfying assignment for the second-order constraints. Throughout this section, we will illustrate the reductions for the constraints in Figure 2.1(c).

For simple examples, fixed-point based techniques like abstract interpretation can be used to discover the unknown invariants I_π. Recently, for the case of conjunctive invariants, use of Farkas' Lemma has been proposed [62] to remove universal quantifiers from the verification condition in Eq. 2.1 to yield a tractable system of constraints. From basic linear programming we know:

Lemma 2.1 (Farkas' Lemma [105, 237]) *A satisfiable system of linear inequalities $\wedge_i e_i \geq 0$ implies an inequality $e \geq 0$ if and only if there exists a non-negative λ_0 and λ_i's such that $\lambda_0 + \sum_i \lambda_i e_i = e$.*

The novelty of our constraint solving approach is three-fold. We first assume *invariant templates* (possibly disjunctive) and then we reduce the program verification condition (possibly involving disjunctions) to unsatisfiability constraints over the parameters of the templates (**Step 1**). We restate and apply Farkas' Lemma in a form suitable for handling unsatisfiability constraints (**Step 2**). Instead of using specialized mathematical solvers [62, 75], we use bit-vector modeling to reduce the constraints to SAT formulae that can be solved using off-the-shelf SAT solvers (**Step 3**). Despite having disjunctive templates, the constraint formulae generated for program verification are conjunctive. This will not be the case for more sophisticated analyses, as we will see later.

Step 1 First, we convert second-order unknowns to first-order unknowns. Instead of searching for a solution to unknown relations (which are second-order entities) from an arbitrary domain, we restrict the search to a template that is some boolean combination of linear inequalities among program variables. For example, an unknown relation can have the template $(\sum_i a_i x_i \geq 0 \wedge \sum_i b_i x_i \geq 0) \vee (\sum_i c_i x_i \geq 0 \wedge \sum_i d_i x_i \geq 0)$, where a_i, b_i, c_i, d_i are all unknown integer constants and x_i are the program variables. The template can either be provided by the user (for example, by specifying the maximum number of conjuncts and disjuncts in DNF representation of any unknown relation), or we can have an iterative scheme in which we progressively increase the size of the template until a solution is found. Given such templates, we replace the unknown relations in the constraint in Eq. 2.1 by the templates and then apply any substitutions present in the verification condition, to obtain a first-order logic formula with unknowns that range over integers.

For the example in Figure 2.1(a), a relevant invariant template is $a_1x + a_2y + a_3 \geq 0 \vee a_4x + a_5y + a_6 \geq 0$, where the a_i's are (integer) unknowns to be discovered. If the chosen domain for the template is not expressive enough then the constraints will be unsatisfiable. On the other hand if there is redundancy then redundant templates can always be instantiated with `true` or `false` as required. This step of the reduction translates the verification condition in Figure 2.1(c) with unknown I to unknowns a_i's, e.g. the first constraint in Figure 2.1(c) after **Step 1** is `true` $\Rightarrow (-50a_1 + a_2y + a_3 \geq 0) \vee (-50a_4 + a_5y + a_6 \geq 0)$.

Step 2 Next, we translate first-order universal to first-order existential quantification using Farkas' Lemma (at the cost of doing away with some integral reasoning). Farkas' Lemma implies that a conjunction of linear inequalities $e_i \geq 0$ (with integral coefficients) is unsatisfiable over reals iff some non-negative (integral) linear combination of e_i yields a negative quantity, i.e.,

$$\forall X \left(\neg (\bigwedge_i e_i \geq 0) \right) \iff \exists \lambda > 0, \lambda_i \geq 0 \left[\forall X \left(\sum_i \lambda_i e_i \equiv -\lambda \right) \right]$$

The reverse direction of the above lemma is easy to see since it is not possible for a non-negative linear combination of non-negative expressions e_i to yield a negative quantity. The forward direction also holds since the only way to reason about linear inequalities over reals is to add them, multiply them by a non-negative quantity, or add a non-negative quantity.

The universal quantification in the right hand side of the above equivalence is over a polynomial equality, and hence can be eliminated by equating the coefficients of the program variables X on both sides of the polynomial equality.

We can convert any universally quantified linear arithmetic formula $\forall X(\phi)$ into an existentially quantified formula using Farkas' Lemma as follows. We convert ϕ to conjunctive normal form $\bigwedge_i \phi_i$, where each conjunct ϕ_i is a disjunction of inequalities $\bigvee_j e_i^j \geq 0$. Observe that $\forall X(\phi) = \bigwedge_i \forall X(\phi_i)$ and that ϕ_i can be rewritten as $\neg \bigwedge_j (-e_i^j - 1 \geq 0)$. Hence, Farkas' Lemma, as stated above, can be applied to each $\forall X(\phi_i)$.

We illustrate the application of this step over the first constraint from Figure 2.1(c), which we obtained after **Step 1**. After **Step 1** we have `true` $\Rightarrow$ $e_1 \geq 0 \vee e_2 \geq 0$ (where $e_1 \equiv -50a_1 + a_2 y + a_3 \geq 0$ and $e_2 \equiv -50a_4 + a_5 y + a_6 \geq 0$ as obtained earlier). After expanding the implication we get a constraint that is already in CNF form, and therefore the corresponding unsatisfiability constraint is $\neg((-e_1 - 1 \geq 0) \wedge (-e_2 - 1 \geq 0))$. Farkas' Lemma can now be applied to yield $\exists \lambda_1, \lambda_2 \geq 0, \lambda > 0 (\forall_{x,y} \lambda_1(-e_1 - 1) + \lambda_2(-e_2 - 1) \equiv -\lambda)$. Now we can collect the coefficients for x, y to get a first-order existential constraint. Notice that λ_1 (respectively λ_2) is multiplied with the coefficients inside e_1 (respectively e_2), and therefore this is a multi-linear quadratic constraint over integers. Equating the coefficients of x, y and the constant term we get the constraints: $(50a_1\lambda_1 - a_3\lambda_1 - \lambda_1) + (50a_4\lambda_2 - a_6\lambda_2 - \lambda_2) = -\lambda$ and $a_2\lambda_1 + a_5\lambda_2 = 0$.

Farkas' Lemma applies to reals and its application leads to a loss of completeness as we do away with *integral reasoning.* For example, Farkas' Lemma cannot help us prove unsatisfiability of $3x \geq 1 \wedge 2x \leq 1$ with x ranging over integers. Farkas' Lemma would check that there exist satisfying values for x, namely $\frac{1}{3} \leq x \leq \frac{1}{2}$. While there is a discrete version of Farkas' Lemma [181], it involves solving an explicit linear programming problem of fixed dimension, and we find the added complexity too expensive. We find that this loss of completeness in using the real version of Farkas' Lemma is not a hindrance in any of our examples.

Step 3 Next, we convert first-order existential (or quantifier-free) to SAT. The formulas that we obtain from the above step are (multi-linear quadratic polynomials) over integer variables. We convert these formulas into SAT formulas by modeling integer variables as bit vectors and encode integer operations like arithmetic, multiplication, and comparison as boolean operations over bit vectors.

Properties of satisfiability-based invariant generation Our approach to constraint solving is sound in the sense that any satisfying solution to the SAT formula yields a valid proof of correctness. However, it is not complete, i.e., there might exist a valid proof of correctness but the SAT formula might not be satisfiable. This is expected since program verification in general is an undecidable problem, and no algorithm can be both sound and complete. These properties are formalized by the following theorem.

Theorem 2.1 (Soundness and Relative Completeness) *Let an inductive invariant exist that proves the program assertions, and let $\phi_T(vc)$ be the SAT formula generated using* **Steps 1,2**, *and* **3** *over the verification condition vc using a template specification T (defined in Section 2.2.2). Then, any satisfying solution to $\phi_T(vc)$ corresponds to an inductive invariant (soundness), and $\phi_T(vc)$ is satisfiable (relative completeness) as long as:*

1. *An inductive invariant exists as an instantiation of the template specification T, i.e., we can get the invariant by instantiating the coefficients in $template(T)$ using integers representable using bit vectors of maximum size $bvsize(T)$.*

2. *Every implication in vc can be discharged without using properties of integers, i.e., without integral reasoning.*

Proof: From the soundness and completeness (up to termination) of verification condition generation [92, 266] we know that if an inductive invariant exists, it will be a solution to the verification condition vc constructed using Eq. 2.1. We just need to ensure that $\phi_T(vc)$ has the same solutions, up to difference in representation, as vc. We prove each direction in turn:

- *Soundness* If $\phi_T(vc)$ has a satisfying boolean solution, then from the soundness of the bit-vector encoding in **Step 3**, we know that it corresponds to an integral solution to the linear equations after **Step 2**. By Farkas' Lemma, we know that a satisfying solution to λ_0, λ_i's, and the invariants exists only if the vc implications are satisfied when we substitute the invariant in them. Given that we have a satisfying solution it means that the solution is also a solution to *vc*.

- *Relative completeness* If an inductive invariant exists then it has to be a solution to *vc*. By assumption *1* above, the invariant is an instantiation of *template*(T). Therefore after the substitution in **Step 1** the constraints have the same set of satisfying solutions as *vc*. By Farkas' Lemma, we know that the integer constraints after **Step 2** have a satisfying solution if *vc* has a satisfying solution, i.e., the invariant. By assumption *2* above, we also know that no property of integers over reals is required and consequently, **Step 2** retains all satisfying solutions. By assumption *1* above, we know that each integer coefficient in the invariant can be represented using *bvsize*(T) bits and consequently the bit-vector encoding of **Step 3** retains all solutions as well. (We assume that the λ's required are sufficiently small, i.e., their absolute values are less than $2^{bvsize(T)-1}$, so that they can be encoded safely too. If this assumption is not valid then they can be chosen to be of arbitrarily large size.) Thus *vc* is satisfiable only if $\phi_T(vc)$ is satisfiable under the assumption *1* and *2* above.

□

We have found that the completeness assumptions do not hinder invariant inference in practice. The right templates are easily found by iterative guessing, easily mechanized if required, and most programs stick to reasoning that is equally valid over reals as over integers. The real challenge instead lies in finding the satisfiability assignment for the SAT formula, for which the recent engineering advances in SAT solvers seem to stand up to the task.

2.2.4 Choice of cut-set

The choice of a cut-set affects the precision and efficiency of our algorithm—and in fact, of any other technique with similar objectives. We find that the choice of a cut-set has significant bearing on expressivity but has been seriously under-studied. A recent proposal [30] performs fixed-point computation on top of a constraint-based technique to regain precision, which we claim was lost in the first place because of a non-optimal choice of cut-set. In this section, we describe a strategy for choosing a cut-set that strikes a good balance between precision and efficiency.

From the definition of a cut-set, we know that we need to include some program locations from each loop into the cut-set. A simple choice for the cut-set includes all header nodes (or targets of back-edges) as cut-points, and is the typical approach. This cut-set, which we will refer to as C_{head}, necessitates searching/solving for unknown relations over disjunctive relations when the proof of correctness involves a disjunctive loop invariant. It is interesting to note that for several programs that require disjunctive loop invariants, there is another choice for cut-set that requires searching for unknown relations with fewer disjuncts, or even only conjunctive.

This expressive cut-set $C_{precise}$ that minimizes disjunctive relations corresponds to choosing one cut-point on each disjoint path inside the loop. Notice that such a choice may not correspond to any assignment of cut-points to syntactic program locations. Consider the case of multiple conditionals in sequence inside a loop, in which case in the proof, which refers to the cut-points, we need to expand the control flow inside the loop. For example, two conditionals in sequence give rise to four cut-points corresponding to the four disjoint paths, but only when the control flow is expanded can these four points be identified. This cut-set leads to the greatest precision in the following sense.

Theorem 2.2 (Best cut-set) *Let $C_{precise}$ be a cut-set that includes a cut-point on each acyclic path inside a loop (after expansion of control flow into disjoint paths). For invariants within a given template specification T (with arbitrarily large coefficients as required), if there exists a solution for some cut-set, then there exists a solution for $C_{precise}$.*

> Proof: Suppose there exists a solution to the relations (of a specified boolean structure) in some cut-set C'. We show that a solution will exist in the cut-set $C_{precise}$. Let p_i be the disjoint paths inside the loop (for the cut-set $C_{precise}$) and p'_i be the disjoint paths on which the unknown relations I'_i are found for cut-set C'. Notice that in C' there may be more than one cut-point on each path. As mentioned earlier, for an acyclic path, a relation at any point on the path can be easily translated to any other point, and therefore we ignore multiple relations on the same path. Also, by the definition of a cut-set each path through the loop has to have a cut-point.
>
> We construct a solution to the relations in $C_{precise}$ as follows: For each disjoint path p_i which has a relation I'_i in C' we assign the relation I'_i. For paths p_i and

p_j that are disjoint in $C_{precise}$ but treated as a single path p_{ij} with invariant I'_{ij} in C' we assign the same relation I'_{ij} to both paths. It is trivial to see that this invariant will be a valid one. Therefore, there exists a solution for the cut-set $C_{precise}$.

□

Furthermore, there are several examples that show that the reverse direction in Theorem 2.2 is not true, i.e., for a given template structure for the unknown relations, there exists a solution with cut-set $C_{precise}$ but there is no solution with other choices of cut-sets. This is illustrated by the example in Figure 2.2 and discussed in Section 2.2.5.

While choosing $C_{precise}$ does give us the most expressivity for a given template specification, it also inserts the most number of unknown relations, which can be expensive to infer. The cut-set C_{head} is at the other end least expensive and least expressive in this regard. Therefore we balance expressivity and expensiveness by picking cut-sets between the two extremes of $C_{precise}$ and C_{head}, experimentally.

Experimental heuristic strategy for choosing cut-set If the loop body has few conditionals in sequence, then we choose the strategy which has the best chance of yielding a proof of correctness over a fixed unknown invariant template, as described in Theorem 2.2. However, this scheme can be costly if the loop body has several sequential conditionals since the number of acyclic paths inside the loop is exponential in the number of sequential conditionals inside the loop. Hence, in such a case we choose

```
   PV1() {
1    $x := 0$; $y := 0$;
2    while (true) {
3      if ($x \leq 50$)
4        $y$++;
5      else
6        $y$--;
7      if ($y < 0$)
8        break;
9      $x$++;
10   }
11   assert($x = 102$)
   }
```

Figure 2.2: Program verification example, from work on widening techniques by Gopan and Reps [125], that requires a disjunctive invariant at the loop header. But a clever choice of cut-set leads to conjunctive invariants.

multiple join points inside the loop, each separated by a few conditionals, as the cut-points.

2.2.5 Examples

Consider the example shown in Figure 2.2. Let π_i denote the program point that *immediately precedes* the statement at line i in the program. The simplest choice of cutpoint corresponds to the loop header at π_2. The inductive invariant that is needed, and is discovered by our tool, is the disjunction $(0 \leq x \leq 51 \wedge x = y) \vee (x \geq 51 \wedge y \geq 0 \wedge x + y = 102)$. Typically programs work in phases [125] and the disjunctions in the invariants have predicates from the conditionals that split the phases. Notice that they are also syntactically differentiable in terms of the disjoint paths inside the loop.

Conjunctive invariants are easier to discover, and we now show how such programs can be handled more efficiently by discovering a set of conjunctive invariants instead of a single disjunctive one. In particular, if the cut-set was chosen to be $\{\pi_4, \pi_6\}$ then the inductive invariant map is indeed conjunctive. Our algorithm discovers the inductive invariant map $\{\pi_4 \mapsto (y \geq 0 \wedge x \leq 50 \wedge x = y), \pi_6 \mapsto (y \geq 0 \wedge x \geq 50 \wedge x + y = 102)\}$. We can verify that this invariant map is indeed inductive. The interesting cases are for paths starting from π_4 and ending at $\{\pi_4, \pi_6\}$. It is trivial to verify the path that ends at the same location. The path from π_4 to π_6 is non-trivial only for the case during the transition between the phases, which happens when $x = y = 50$ at π_4 and therefore $x = y = 51$ at π_6. For this program the meaningful paths starting from π_6 only end at the same location because the program does not alternate between phases. But if it did then a case similar to π_4 would arise.

A wide variety of techniques based on fixed point computation and CFG elaboration [125, 30, 232] exist for the programs whose invariants lend themselves to such partitioning, and therefore it is no surprise that they can be efficiently handled using our cut-set optimization. We go further by not committing ourselves to conjunctive invariants for the individual phases. If some phase of the program was more complicated, possibly requiring disjunctions itself, then even the best choice of the cut-set would leave some disjunctive invariants to be discovered. Our technique is not constrained to handle just conjunctive invariants. Disjunctive invariants, which are very difficult to discover using previous approaches, are easily found in our framework.

The example in Figure 2.1(a) has no phases and no conditionals inside the loop, and the only inductive invariant describing the loop, $x < 0 \vee y > 0$, is disjunctive and is discovered by our technique. Heuristic proposals for handling disjunction [232, 30] will fail to efficiently discover invariants for such programs.

2.3 Interprocedural Analysis

The ω computation described in previous section is applicable only in an intraprocedural setting. In this section, we show how to extend our satisfiability-based technique to precise (i.e., context-sensitive) interprocedural analysis.

Precise interprocedural analysis is challenging because the behavior of the procedures needs to be analyzed in a potentially unbounded number of calling contexts. Procedure inlining is one way to do precise interprocedural analysis. However, there are two problems with this approach. First, procedure inlining may not be possible at all in presence of recursive procedures. Second, even if there are no recursive procedures, procedure inlining may result in an exponential blowup of the program. For example, if procedure P_1 calls procedure P_2 twice and procedure P_2 calls procedure P_3 twice, then procedure inlining would result in four copies of procedure P_3 inside procedure P_1. In general, leaf procedures can be replicated an exponential number of times.

A more standard way to do precise interprocedural analysis is to compute procedure summaries, which are relations between procedure inputs and outputs. More specifically, these summaries are usually structured as sets of pre/postcondition pairs (A_i, B_i), where A_i is some relation over procedure inputs and B_i is some relation over procedure inputs and outputs. The pre/postcondition pair (A_i, B_i) denotes that whenever the procedure is invoked in a calling context that satisfies constraint A_i, the procedure ensures that the outputs will satisfy the constraint B_i. However, there is no automatic recipe to efficiently construct or even represent these procedure summaries, and abstraction-specific techniques may be required. Data structures and algorithms for representing and computing procedure summaries have been described over the abstractions of linear constants [231] and linear equalities [204]. Recently, some heuristics have been described for the abstraction of linear inequalities [238].

In this section, we show the satisfiability-based approach is particularly suited to discovering such useful pre/postcondition (A_i, B_i) pairs. The key idea is to observe that the desired behavior of most procedures can be captured by a small number of such (unknown) pre/postcondition pairs. We then replace the procedure calls by these unknown behaviors and assert that the procedure has such behaviors, as in assume-guarantee style reasoning. Assume-guarantee reasoning has been used for modular reasoning [159, 216] about components of a program under assumptions that the components make about their environment. These assumptions are then discharged when modularly reasoning about other components that use it.

For ease of presentation and without loss of generality, let us assume that a procedure does not read/modify any global variables; instead all global variables that are read by the procedure are passed in as inputs, and all global variables that are modified by the procedure are returned as outputs. Our tool $\texttt{VS}^3_{\texttt{LIA}}$ does this automatically and can handle globals seamlessly. We now describe the steps of our interprocedural analysis algorithm.

We first assume that there are q interesting calling contexts for procedure $P(\vec{x})\{S; \texttt{return } \vec{y};\}$ with the vector of formal arguments $\vec{x}$ and vector of return values $\vec{y}$. The value of q can be iteratively increased until invariants are found that make the constraint system satisfiable. Then, we summarize the behavior of each procedure using q tuples (A_i, B_i) for $1 \leq i \leq q$, where A_i is some relation over procedure inputs $\vec{x}$, while B_i is some relation over procedure inputs and outputs $\vec{x}$ and $\vec{y}$. We assert that this is indeed the case by generating constraints for each i as below and asserting their conjunction:

$$\texttt{assume}(A_i);\ S;\ \texttt{assert}(B_i) \tag{2.2}$$

We compile away procedure calls $\vec{v} := P(\vec{u})$ on any simple path by replacing them with the following code fragment:

$$\vec{v} :=?; \texttt{assume}\left(\bigwedge_i (A_i[\vec{u}/\vec{x}] \Rightarrow B_i[\vec{u}/\vec{x}, \vec{v}/\vec{y}])\right); \tag{2.3}$$

The correctness of this encoding follows directly from the correctness of tabulation-based procedure summary computation [73], i.e., summaries that explicitly state an abstract relations on the inputs as output, as studied for dataflow analysis over finite lattices [241], and even for some infinite domains [226]. In this section, we have considered abstract, but explicit, pre- and postcondition facts, unlike some previous approaches [134, 271] that use symbolic constants to generalize the summaries. The advantage of our approach here is that it is goal-oriented, and computes only those facts in the summary that are required for the analysis of call locations. Such a luxury was not afforded by previous dataflow approximation techniques, which had to compute the most precise facts because they either analyzed in a forwards or backwards direction, but not both. We will revisit summary computation again in Section 2.6 where we attempt to compute the most precise summaries possible.

Observe that in our approach, there is no need, in theory, to have q different pre/postcondition pairs. In fact, the summary of a procedure can also be represented as some formula $\phi(\vec{x}, \vec{y})$ (with arbitrary Boolean structure) that represents a relation between procedure inputs $\vec{x}$ and outputs $\vec{y}$. In such a case, we assert that ϕ indeed is the summary of procedure P by generating constraint for $\{S; \texttt{assert}(\phi(\vec{x}, \vec{y}))\}$, and we compile away a procedure call $\vec{v} := P(\vec{u})$ by replacing it by the code fragment $\vec{v} :=?; \texttt{assume}(\phi[\vec{u}/\vec{x}, \vec{v}/\vec{y}])$.

However, in practice, our approach of maintaining symbolic pre/post pairs (which is also inspired by the data structures used by the traditional fixed-point computation algorithms) is more efficient since it enforces more structure on the assume-guarantee proof and leads to fewer unknown quantities and simpler constraints. In particular, by assuming a template for A_i that is only in terms of the procedure inputs, we ensure that the solver cannot prove $\neg A_i$ at the beginning of the procedure. (Otherwise along-with $\texttt{assume}(A_i)$ in Equation (2.2) it could prove $\texttt{false}$, and any arbitrary consequence B_i would follow.)

Optimization If there are a small number q_{small} of static procedure calls, then we can replace the i^{th} procedure call $\vec{v} := P(\vec{u})$ by

$$\texttt{assert}(A_i[\vec{u}/\vec{x}]); \vec{v} :=?; \texttt{assume}(B_i[\vec{u}/\vec{x}, \vec{v}/\vec{y}])$$

where $1 \leq i \leq q_{small}$. This approach is somewhat akin to inlining, as each i^{th} calling context's behavior is encoded by a separate (A_i, B_i), while being able to handle

```
IP1() {
  $x := 5$;  $y := 3$;
  result := Add($x, y$);
  assert(result = 8);
}
Add(int $i$, $j$) {
  if $i \leq 0$
     ret := $j$;
  else
     $b := i - 1$;
     $c := j + 1$;
     ret := Add($b, c$);
  return ret;
}
```

(a)

```
IP2() {
  result :=M(19)+M(119);
  assert(result = 200);
}
M(int $n$) {
  if($n > 100$)
    return $n - 10$;
  else
    return M(M($n + 11$));
}
```

(b)

Figure 2.3: Interprocedural analysis examples (a) taken from previous approaches to summary computation [238, 205] (b) McCarthy 91 function [193, 192, 189] requires multiple summaries.

recursion (if the recursive call can be succinctly described using some (A_k, B_k)). Also, note that there is loss of context-sensitivity in this approach, as syntactic call locations are assumed to be describable using a single summary. For instance, consider a call inside a loop whose behavior is dependent on the loop iterator. This optimization will fail to verify such behavior, while the unoptimized encoding will work. So while this approach may be more efficient for certain cases, in general, we do not use it.

Examples Consider the example shown in Figure 2.3(a). Our algorithm verifies the assertion by generating the summary $i \geq 0 \Rightarrow \texttt{ret} = i + j$ for procedure Add. This example illustrates that only relevant summaries are computed for each procedure. In addition to serving as the base case of the recursion the true branch of the condition inside Add has the concrete effect $i < 0 \Rightarrow \texttt{ret} = j$. But this behavior is not needed to prove any assertion in the program and is therefore automatically suppressed (in that the tool proves the assertions without it) by our goal-oriented summary computation. This example illustrates that our tool finds *any* summary, not necessarily the weakest, for a procedure that is useful for proving program assertions.

The procedure $\texttt{M}(\texttt{int}\ n)$ in Figure 2.3(b) is the McCarthy91 function—proposed by McCarthy, Manna and Pnueli [192, 193, 191] as a challenge problem in recursive program verification—which can be precisely described by the summaries $n > 100 \Rightarrow \texttt{ret} = n - 10$ and $n \leq 100 \Rightarrow \texttt{ret} = 91$. The function has often been used as a benchmark test for automated program verification. The goal-directed nature of the verification problem allows our analyzer to derive $n = 119 \Rightarrow \texttt{ret} = n-10$ and $n \leq 100 \Rightarrow \texttt{ret} = 91$ as the summary, which proves the program assertion. As such, the tool discovers only as much as is required for the proof. For the summary with the antecedent $n \leq 100$ no such simplification exists, and the tool discovers the most precise consequence such that the invariant is inductive.

Consider the example shown in Figure 2.4(a). The assertion in the program needs to be verified for timing/bounds analysis of the quicksort procedure (Section 2.7.1). G is a global variable that is incremented every time the function is called. For each procedure call, $G_{\texttt{in}}$ and $G_{\texttt{out}}$ refer to the value of the global before and after the procedure call, respectively. Our algorithm generates the summary $l - r \leq 1 \Rightarrow G_{\texttt{out}} - G_{\texttt{in}} \leq 2(r - l) + 3$ for the procedure QSort.

Consider the example shown in Figure 2.4(b), which contains a potential infinite recursive call inside F, and also swaps the value stored in y_2 and y_3 between

```
int $G$;
IP3(int $n$) {
  assume($n \geq 1$);
  $G := 0$;
  QSort($0, n$);
  assert($G \leq 2n + 3$);
}
QSort(int $l, r$) {
  $G$++;
  if ($r > l$)
      assume($l \leq m \leq r$);
      QSort($l, m - 1$);
      QSort($m + 1, r$);
}
```

(a)

```
IP4(int $x_1$, $x_2$) {
  $x_3 := 3 \times x_2 - 2$;
  $x_1 := F(x_1, x_2, x_3)$;
  assert($x_1 = 3x_2 - 2$);
}
F(int $y_1$, $y_2$, $y_3$) {
  if (*)
     ret $:= 3 \times y_2 - 2$;
  else
     ret $:= F(y_1, y_3, y_2)$;
  return ret;
}
```

(b)

Figure 2.4: Context-sensitive interprocedural analysis examples (a) over recursive functions [238] and (b) possibly non-terminating function [205].

calls. Thus an iterative refinement scheme that recursively analyses sub-procedures may not terminate. On the other hand, our algorithm verifies that *if the procedure terminates*, then it output values that satisfy the program assertions. (If the procedure does not terminate then the assertion is trivially satisfied.) Our tool $\texttt{VS}^3_{\texttt{LIA}}$ generates the summary $y_3 = 3y_2 - 2 \Rightarrow \texttt{ret} = y_3$ for procedure F which verifies the assertion.

Corollary 2.1 *If there exist q summaries $(A_i, B_i)_{i=1..q}$ with which the assertions in the program are verified, then our encoding generates a SAT instance whose solution corresponds to the q summaries.*

PROOF: The proof is a direct consequence of the soundness of our constraint encoding (Theorem 2.1) and the soundness of our interprocedural tabulation-based summary computation [73, 241, 226].

□

2.4 Maximally weak Precondition

Given a program along with some assertions, the problem of weakest precondition generation is to infer the weakest precondition $I_{\pi_{\texttt{entry}}}$ that ensures that whenever the program is run in a state that satisfies $I_{\pi_{\texttt{entry}}}$, the assertions in the program hold. In Section 2.7 we show that a solution to this problem will be a powerful tool for a wide range of applications.

In this section, we present a satisfiability-based approach for inferring an approximation to weakest preconditions under a given template. Since a precise solution to this problem is undecidable, we work with a relaxed notion of weakest precondition, namely *maximally weak precondition*. For a given template structure T (as described in Section 2.2.3 for invariants), we say that A is a maximally weak precondition if A is an instantiation of T, and there is no valid precondition proving the program assertions that is comparable to and weaker than A within the same template.

The first step to a satisfiability-based approach to maximally weak preconditions is to treat the precondition $I_{\pi_{\texttt{entry}}}$ as an unknown relation in Eq. 2.1. This is unlike program verification, where we set $I_{\pi_{\texttt{entry}}}$ to be `true`. However, this change merely encodes that any consistent assignment to $I_{\pi_{\texttt{entry}}}$ is a valid precondition, not necessarily the weakest or maximally weak one. In fact, when we run our tool with this change, it returns `false`, which is always a valid precondition, as a solution for $I_{\pi_{\texttt{entry}}}$.

One approach to finding the maximally weak precondition may be to search for a precondition that is strictly weaker than the current solution (by adding a weakness constraint to Eq. 2.1) and to iterate until no such precondition exists. However, in practice this approach make slow progress. For Figure 2.5(a), which we discuss below, this technique iteratively produced $i \geq j + 127$, $i \geq j + 126$, $\ldots$, $i \geq j$ as preconditions, under a modeling that used 8-bit two's-complement integers. In general this naïve iterative technique will be infeasible. We need to augment the constraint system to encode the notion of a maximally weak relation.

We can encode that $I_{\pi_{\texttt{entry}}}$ is a maximally weak precondition as follows. The verification condition in Eq. 2.1 can be regarded as function of two arguments $I_{\pi_{\texttt{entry}}}$ and $I_{\texttt{r}}$, where $I_{\texttt{r}}$ denotes the relations at all cut-points except at the program entry location, and can thus be written as $\forall X.\phi(I_{\pi_{\texttt{entry}}}, I_{\texttt{r}})$. Now, for any other relation I' that is strictly weaker than $I_{\pi_{\texttt{entry}}}$, it should be the case that I' is not a valid precondition. This can be stated as the following constraint.

$$
\begin{aligned}
&\forall X.\phi(I_{\pi_{\texttt{entry}}}, I_{\texttt{r}}) \quad \wedge \\
&\qquad\qquad \forall I', I'_{\texttt{r}} \left(\texttt{weaker}(I_{\pi_{\texttt{entry}}}, I') \Rightarrow \neg\forall X.\phi(I', I'_{\texttt{r}})\right)
\end{aligned}
$$

where $\texttt{weaker}(I_{\pi_{\texttt{entry}}}, I') \overset{\texttt{def}}{=} (\forall X.(I_{\pi_{\texttt{entry}}} \Rightarrow I') \wedge \exists X.(I' \wedge \neg I_{\pi_{\texttt{entry}}}))$.

The trick of using Farkas' Lemma to get rid of universal quantification (**Step 2** in Section 2.2.3) cannot be applied here because there is existential quantification nested inside universal quantification. We now consider examples of maximally weak preconditions that we expect to—and indeed do—infer. In the following section we will describe our novel iterative approach to maximally weak precondition inference.

```
WP1(int $i$, $j$) {
  $x := y := 0$;
  while ($x \leq 100$) {
      $x := x + i$;
      $y := y + j$;
  }
  assert($x \geq y$)
}
```

(a)

```
Merge(int $m_1, m_2, m_3$) {
  assert($m_1, m_2 \geq 0$)
  $k := i := 0$;
  while ($i < m_1$) {
     assert($0 \leq k < m_3$)
     $A[k{+}{+}] = B[i{+}{+}]$;
  }
  $i := 0$;
  while ($i < m_2$) {
     assert($0 \leq k < m_3$)
     $A[k{+}{+}] = C[i{+}{+}]$;
  }
}
```

(b)

Figure 2.5: Maximally weak precondition examples.

Examples For the procedure in Figure 2.5(a), our algorithm generates two different preconditions that individually ensure that the program assertion holds: (i) $(i \geq j)$ ensures that if the loop terminates then $x \geq y$, and (ii) $(i \leq 0)$ ensures that the loop never terminates making the assertion unreachable and therefore trivially true.

Notice how each of these preconditions is maximally weak by themselves. For instance, while $i \geq j$ is a valid precondition, $i \geq j - 1$, which is strictly weaker, is not. Additionally, $i \geq j$ and $i \leq 0$ are incomparable to each other. The true weakest precondition is the disjunction of all incomparable maximally weak preconditions.

Figure 2.5(b) shows an array merge function that is called to merge two arrays B and C of sizes m_1 and m_2, respectively, into a third one A of size m_3. The program is correct if no invalid array accesses are made (stated as the assertions inside the loops) when it is run in an environment where the input arrays are proper $(m_1, m_2 \geq 0)$. Our algorithm generates maximally weak preconditions $m_3 \geq m_1+m_2$ and $m_1 = 0 \wedge m_2 = 0$—which are orthogonal to each other.

Notice that we have specified $m_1, m_2 \geq 0$ as an assertion instead of an assumption. This is required because otherwise the tool generates preconditions (e.g., $m_1 < 0$) that, along with the assumption, imply `false` at the beginning of the procedure. To circumvent these trivial cases we need to ensure that all our required assumes appear in the generated precondition, which occurs if they are asserted.

2.4.1 Locally pointwise-weakest strategy

For simplicity of presentation, we assume that each non-trivial maximal strongly connected component in the control flow graph has exactly one cut-point—an assumption that can also be ensured by simple transformations[2]. However, the results in this section can be extended to the general setting without this assumption.

Towards a technique for maximally weak preconditions, we define two characterizations of relations. First is a *pointwise-weakest relation* that connects a relation to relations "spatially" adjacent to it in the control flow graph. The second is a *locally pointwise-weakest relation* that connects a relation to relations "semantically" adjacent to it in the proof lattice. The notion of nearby relations is in different realms for pointwise-weakest and for locally pointwise-weakest. For pointwise-weakest, the

[2] First, merge the targets of back-edges of each maximally strongly-connected component and introduce a special control variable to direct the control flow appropriately. This ensures that it is appropriate to choose the target of the new back-edge as the only cut-point for the entire SCC. Second, map the templates at the original choice of cut-points in the original strongly connected component to one new template at the target of the new single back-edge using backward symbolic execution.

concept of a neighboring relation is a relation at a neighboring, specifically successor, cut-point in the control flow graph. On the other hand, for locally pointwise-weakest, it is the neighbors in a poset lattice ordered by the implication relation.

Definition 2.1 (Pointwise-weakest relations) *A relation I at any cut-point is pointwise-weakest if it is a weakest relation that is consistent with respect to the relations at its successor cut-points.*

Pointwise-weakest relations ensure that when going from one cut-point to another the relations are as maximally weak as possible. Next, we define a notion of weakness with respect to the proof lattice and which ensures that we always consider the weakest relation amongst relations in the “proof neighborhood” of each other. Later, we define a suitable neighborhood $\mathtt{N}$ in the lattice of linear relations.

Definition 2.2 (Locally pointwise-weakest relations) *A relation I is a locally pointwise-weakest with respect to a neighborhood $\mathtt{N}$ if it is a weakest relation among its neighbors that is consistent with respect to the relations at its neighboring—successor—cut-points.*

Our technique for maximally weak preconditions will consist of reducing the problem to finding pointwise-weakest relations, which will in turn reduce to finding locally pointwise-weakest relations. Pointwise weakest relations ensure that (spatial) neighbors are optimally assigned, while locally pointwise-weakest relations ensure that the values at each cut-point are the (semantically) weakest. First, the weakest precondition can be derived from *pointwise-weakest* relations at each cut-point in

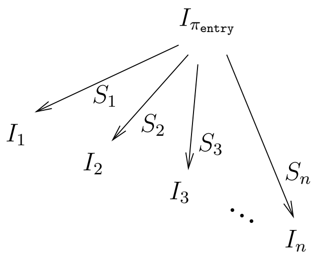


Figure 2.6: Maximally weak preconditions as pointwise-weakest relations.

reverse topological order of the control dependences between different cut-points. Note that since we assume that each maximal SCC has at most one cut-point, there are no cyclic control dependencies between different cut-points. Second, a pointwise-weakest relation can be derived from *locally pointwise-weakest* relation and repeating the process to obtain a weaker locally pointwise-weakest relation if one exists. Intuitively, this second iteration steps through local minimas to reach the global minima.

Theorem 2.3 (Maximally weak preconditions) *A precondition is maximally weak if it is a pointwise-weakest relation at the program entry point, and every other relation in the program is also pointwise-weakest.*

PROOF: Suppose otherwise that a precondition $I_{\pi_{\texttt{entry}}}$ is not maximally weak while it is the case that all relations, including the precondition, are pointwise-weakest. Since $I_{\pi_{\texttt{entry}}}$ is not maximally weak, we can construct another I' such that it is comparable and strictly weaker than it, i.e., $\texttt{weaker}(I_{\pi_{\texttt{entry}}}, I')$ holds. Consider the set of relation $\{I_i\}_{i=1..n}$ at the successor cut-points to the precondition. Figure 2.6 shows the scenario. We know from $I_{\pi_{\texttt{entry}}}$ being pointwise-weakest that $I_{\pi_{\texttt{entry}}}$ is the weakest fact that satisfies $I_{\pi_{\texttt{entry}}} \Rightarrow X$ for all $X \in \{I_i[S_i]\}$. Since

I' is weaker that $I_{\pi_{\texttt{entry}}}$ it implies that some X is weaker, and the corresponding I_i is weaker (since S_i remain identical). Since the original intermediate relations were pointwise-weakest, now at least one of the successors of I_i will have to be correspondingly weaker. Transitively, at least one relation will be required to be weaker, that is also a user provided assertion—which cannot be weaker than specified, hence a contradiction.

□

We now define the proof neighborhood that Definition 2.2 uses.

Definition 2.3 (Neighborhood Structure $\texttt{N}_c$) *We define a set of relations that are in the neighborhood $\texttt{N}_c$ of a conjunctive relation (in which, without loss of generality, all inequalities are independent of each other), with c being the largest constant we allow, as follows:*

$$\begin{array}{rcl} \texttt{N}_c(\bigwedge_i e_i \geq 0) & = & \{e_j + \frac{1}{c} \geq 0 \wedge \bigwedge_{i \neq j} e_i \geq 0 \mid j\} \quad \cup \\ & & \{e_j + \frac{1}{c} e_\ell \geq 0 \wedge \bigwedge_{i \neq j} e_i \geq 0 \mid j \neq \ell\} \end{array} \tag{2.4}$$

Neighborhood structure is computable Notice how a neighborhood structure helps template-based invariant inference by ensuring that $\texttt{N}_c$ is computable even for unknown (template) relations. The unknown relations e_j are of a template form $c_{j,0} + c_{j,1}x + c_{j,2}y + c_{j,3}z \ldots \geq 0$, where $c_{j,i}$'s are constant coefficients less than c, and x, y, z are program variables. Then each term in the set comprehensions in Eq. (2.4) can be obtained as another linear relation, with appropriate unknown linear coefficients obtained by collecting terms. For example, $e_j + \frac{1}{c}e_l$ is another linear relation with combinations of the coefficients of e_j and e_l and expands to $(c_{j,0} + \frac{1}{c}c_{l,0}) + (c_{j,1} + \frac{1}{c}c_{l,1})x + (c_{j,2} + \frac{1}{c}c_{l,2})y + (c_{j,3} + \frac{1}{c}c_{l,3})z \mathinner{..} \geq 0$.

Geometric Interpretation The neighborhood structure $\mathtt{N}_c$ has a nice geometric interpretation. The neighbors of a convex region $\bigwedge_i e_i \geq 0$ are obtained by slightly moving any of the hyper-planes $e_j \geq 0$ parallel to itself, or by slightly rotating any of the hyper-planes $e_j \geq 0$ along its intersection with any other hyper-plane $e_\ell \geq 0$.

We extend the neighborhood structure to relations in DNF form (in which, without loss of generality, all disjuncts are disjoint with each other) as:

$$\mathtt{N}_c(\bigvee_i I_i) \quad = \quad \{I'_j \vee \bigvee_{i \neq j} I_i \mid I'_j \in \mathtt{N}_c(I_j)\}$$

Intuitively, $\mathtt{N}_c(I)$ defines the set of all immediate weaker neighbors of I in the poset of all linear arithmetic formulas involving constants less than c and ordered by implication. This is formalized by the the following lemma:

Lemma 2.2 ($\mathtt{N}_c$ = Immediately weaker neighbors) *For all relations I' that are weaker than I, there is some relation $I'' \in \mathtt{N}_c(I)$ such that $I \Rightarrow I'' \Rightarrow I'$.*

The proof of the this lemma is given in Appendix A, Section A.1, and is used to subsequently prove the following theorem:

Theorem 2.4 *Let π be a program point that does not lie inside any loop. Then, any locally pointwise-weakest relation (with respect to the neighborhood structure $\mathtt{N}_c$) at π is also a pointwise-weakest relation at π.*

Theorem 2.4 tells us that pointwise-weakest relations may be directly obtained from locally pointwise-weakest relations for the case of program points outside of

```
Swap(int $x$) {
  while $(*)$
      if $(x = 1)$
          $x := 2$;
      else if $(x = 2)$
          $x := 1$;
  assert($x \leq 8$);
}
```

Figure 2.7: Illustrating the need for iteration in maximally weak precondition inference. Example that has two local minima $x < 1$ and $x \leq 8$ of which only the latter is the maximally weak precondition.

loops. However, for a program point π inside a loop, a locally pointwise-weakest relation may not be a pointwise-weakest relation, as we illustrate by the following example.

Example 2.2 *Let c be the maximum constant allowed in the system. Then in Figure 2.8(a) the locally pointwise-weakest relation $x \leq 1 - \frac{1}{c}$ is not pointwise-weakest. Of the relations expressible in the system, the closest weaker relation ($x \leq 1$) is not consistent, and therefore $x \leq 1 - \frac{1}{c}$ is locally pointwise-weakest but not pointwise-weakest, as indicated by the the presence of $x \leq 8$. Notice that other relations, e.g., $x \leq 3$, are not locally pointwise-weakest, since their neighborhood contains a consistent relation $x \leq 3 + \frac{1}{c}$.*

Computing maximally weak preconditions in practice In practice, we need to compute maximally weak preconditions for programs that have loops in addition to straight-line fragments. So while Theorem 2.4 allows precise derivation of maximally weak relations for loop free fragments, we may need to iterate over the locally pointwise-weakest relations inside loops. Notice that by ensuring we stick to locally pointwise-weakest relation, in each iteration we will make the largest step to the next point of discontinuity. For instance, for Example 2.2, we will take at most two steps in the iterations, in stepping from $x < 1$ to the final solutions $x \leq 8$. Since the solver's decision is not directed by approximate refinement, it may be the case that it outputs the pointwise-weakest relation in the first iteration, terminating in fewer steps.

Notice that even for cases where the pointwise-weakest relations are discovered without iteration, it is instructive to ask the solve for additional (orthogonal) solutions to ensure that the resulting precondition is as close to the weakest precondition as possible. For instance, suppose the weakest precondition is $\bar{I} \vee I_1 \vee I_2$, and suppose $\bar{I}$ is not expressible in the template while I_1 and I_2 are. Also, let I_1 and I_2 be orthogonal to each other. In this case, we may get I_1 directly, or through iterations over locally pointwise-weakest to eventually get the pointwise-weakest, if a loop is involved. We would still prefer to iterate to get other orthogonal solutions. The solve will be able to generate I_2 and subsequently claim that no other solutions are in the template. At that point we will output $I_1 \vee I_2$ as the approximation to the weakest precondition.

2.5 Maximally strong Postcondition

Given a program with a precondition, typically *true*, the problem of strongest postcondition inference is to generate the most precise invariants at all, or a given set of, cut-points. Typically, we are interested in the strongest postcondition $I_{\pi_{\texttt{exit}}}$ at the program exit. Just as in the weakest precondition case, we work with a relaxed notion of strongest postcondition, namely *maximally strong postconditions*. For a given template structure T (as described in Section 2.2.3 for invariants) we say that A is a maximally strong postcondition if A is an instantiation of T, and there is no postcondition comparable to, and stronger than, A within template T.

We can encode that $I_{\pi_{\texttt{exit}}}$ is a maximally strong postcondition as follows. The verification condition in Eq. 2.1 can be regarded as function of two arguments $I_{\pi_{\texttt{exit}}}$ and $I_{\texttt{r}}$, where $I_{\texttt{r}}$ denote the relations at all cut-points except at the program exit location, and can thus be written as $\forall X.\phi(I_{\pi_{\texttt{exit}}}, I_{\texttt{r}})$. Now, for any other relation I' that is strictly stronger than $I_{\pi_{\texttt{exit}}}$, it should not be the case that I' is a valid postcondition. This can be stated as the following constraint.

$$\begin{array}{l} \forall X.\phi(I_{\pi_{\texttt{exit}}}, I_{\texttt{r}}) \quad \wedge \\ \qquad\qquad \forall I', I'_{\texttt{r}}\, (\texttt{stronger}(I_{\pi_{\texttt{exit}}}, I') \Rightarrow \neg\forall X.\phi(I', I'_{\texttt{r}})) \end{array}$$

where $\texttt{stronger}(I_{\pi_{\texttt{exit}}}, I') \stackrel{\texttt{def}}{=} (\forall X.(I' \Rightarrow I_{\pi_{\texttt{exit}}}) \wedge \exists X.(\neg I' \wedge I_{\pi_{\texttt{exit}}}))$.

Our technique for generating maximally strong postcondition is very similar to the maximally weak precondition technique described in Section 2.4. The key idea is to replace occurrences of constant c in the locally pointwise-weakest strategy (Eq. 2.4) for maximally weak precondition by $-c$ to obtain corresponding strategies for generating maximally strong postconditions. The corresponding neighborhood structure is defined to be:

$$\begin{array}{rcl} \texttt{N}_c(\bigwedge_i e_i \geq 0) & = & \{e_j - \frac{1}{c} \geq 0 \wedge \bigwedge_{i \neq j} e_i \geq 0 \mid j\} \quad \cup \\ & & \{e_j - \frac{1}{c}e_\ell \geq 0 \wedge \bigwedge_{i \neq j} e_i \geq 0 \mid j \neq \ell\} \end{array} \tag{2.5}$$

```
SP2() {
  $d := t := s := 0$;
  while(1)
     if (*)
        $t$++; $s := 0$;
     else if (*)
        if ($s < 5$)
           $d$++; $s$++;
}
```

Figure 2.8: Maximally strong postcondition examples taken from sophisticated widening approaches [125, 126].

Examples To infer the maximally strong postconditions for the example in Figure 2.2 we remove the assertion on line 11 and for generality abstract away the constant (50) as m. Our algorithm generates the postcondition $x = 2m + 2$.

For the procedure in Figure 2.8, our algorithm generates two orthogonal solutions in two iterations: $s + d + t \geq 0$ and $d \leq s + 5t$. Iteratively solving for additional solutions allows us to generate such orthogonal solutions. In each subsequent iteration we augment the original formula with a constraint that ensures the orthogonality of new solutions with respect to already generated ones.

2.6 Specification Inference = Interprocedural + maximally weak preconditions + maximally strong postconditions

With a maximally weak precondition and maximally strong postcondition inference technique at hand, we now revisit the interprocedural analysis from Section 2.3 to define specification inference as augmented summary computation. Given a procedure P, a summary[3] set $\{(A_i, B_i)\}_{1\leq i\leq q}$ is called *precise* and *concise* as follows (formalization of the informal definition proposed earlier [271]):

Definition 2.4 (Precise and concise summaries) *$S = \{(A_i, B_i)\}_{1\leq i\leq q}$, a summary set for P, is*

- Precise *if for any valid summary (A', B') for P there exists some $(A_k, B_k) \in S$ such that $A' \Rightarrow A_k$ and $A' \Rightarrow (B_k \Rightarrow B')$[4]. That is, for every valid summary there exists a summary in S that is at least as good (weaker in the assumptions and stronger in the assurance).*

- Concise *if for any (A', B') that satisfies $A_k \Rightarrow A' \wedge A' \not\Rightarrow A_k$ and $A_k \Rightarrow (B' \Rightarrow B_k)$ for some $(A_k, B_k) \in S$, it is the case that (A', B') is not a valid summary for P. Similarly, if $A_k \Rightarrow A'$ and $A_k \Rightarrow (B' \Rightarrow B_k \wedge B_k \not\Rightarrow B')$ for some $(A_k, B_k) \in S$, it is the case that (A', B') is not a valid summary for P. That is, any summary that is strictly better (either strictly weaker in the assumption or strictly stronger in the assurance) than some summary in S is not a valid summary.*

Example 2.3 *Consider the simple program $P(x, y)\{r := 0; \texttt{while}(x > y)\{r := r + 1; x := x - 1\}; \texttt{return}\ r;$ Then a concise and precise summary set is $\{(x \geq y, \texttt{ret} = x - y), (x < y, \texttt{ret} = 0)\}$.*

[3] As noted before, it is entirely a matter of efficiency that we treat the summary as a pair. (A, B) may very well be treated as a single formula—with a better summary being the one that is stronger.

[4] Notice that the check on the assurance, i.e., $B_1 \Rightarrow B_2$, is made under the current context, i.e., A, and hence the extra assumption, i.e., $A \Rightarrow (B_1 \Rightarrow B_2)$.

A summary set that is precise and concise is correspondingly relevant and efficient. It is relevant because analyzing a call location $\vec{v} := P(\vec{u})$ using a precise summary yields the same outcome as with any other valid summary. It is efficient because a concise summary does not contain any redundant facts. For example, for the case of conjunctive pre- and postconditions in summary (A, B), removing any (independent) conjunct from A, and correspondingly adding any (non-implied) conjunct to B, invalidates the summary. If we can generate concise summaries then they can be extended by iteratively enumerating them to get a precise summary set. The notion of a concise summary is essentially a combination of maximally weak preconditions and maximally strong postconditions.

We can encode that $(I_{\pi_{\texttt{entry}}}, I_{\pi_{\texttt{exit}}})$ is a concise summary as follows. The verification condition in Eq. 2.1 can now be regarded as function of three arguments $I_{\pi_{\texttt{entry}}}, I_{\pi_{\texttt{exit}}}$ and $I_{\texttt{r}}$, where $I_{\texttt{r}}$ denote the relations at all cut-points except at the program entry and exit locations, and can thus be written as $\forall X.\phi(I_{\pi_{\texttt{entry}}}, I_{\pi_{\texttt{exit}}}, I_{\texttt{r}})$. Now, for any other relation $I'_{\pi_{\texttt{entry}}}$ that is strictly weaker than $I_{\pi_{\texttt{entry}}}$, it should not be the case that $I'_{\pi_{\texttt{entry}}}$ is a valid precondition, for a *fixed* $I_{\pi_{\texttt{exit}}}$. Similarly, for any other relation $I'_{\pi_{\texttt{exit}}}$ that is strictly stronger than $I_{\pi_{\texttt{exit}}}$, it should not be the case that $I'_{\pi_{\texttt{exit}}}$ is a valid postcondition, for any *fixed* $I_{\pi_{\texttt{entry}}}$. This can be stated as the following constraint.

$$\begin{aligned}
&\forall X.\phi(I_{\pi_{\texttt{entry}}}, I_{\pi_{\texttt{exit}}}, I_{\texttt{r}})\\
&\qquad\qquad \wedge\ \forall I'_{\pi_{\texttt{entry}}}, I'_{\texttt{r}} \left(\texttt{weaker}(I_{\pi_{\texttt{entry}}}, I'_{\pi_{\texttt{entry}}}) \Rightarrow \neg\forall X.\phi(I'_{\pi_{\texttt{entry}}}, I_{\pi_{\texttt{exit}}}, I'_{\texttt{r}})\right)\\
&\qquad\qquad \wedge\ \forall I'_{\pi_{\texttt{exit}}}, I'_{\texttt{r}} \left(\texttt{stronger}(I_{\pi_{\texttt{exit}}}, I'_{\pi_{\texttt{exit}}}) \Rightarrow \neg\forall X.\phi(I_{\pi_{\texttt{entry}}}, I'_{\pi_{\texttt{exit}}}, I'_{\texttt{r}})\right)
\end{aligned}$$

where as before,

$$\begin{aligned}\texttt{weaker}(I_{\pi_{\texttt{entry}}}, I'_{\pi_{\texttt{entry}}}) &\overset{\texttt{def}}{=} (\forall X.(I_{\pi_{\texttt{entry}}} \Rightarrow I'_{\pi_{\texttt{entry}}}) \wedge \exists X.(I'_{\pi_{\texttt{entry}}} \wedge \neg I_{\pi_{\texttt{entry}}}))\\ \texttt{stronger}(I_{\pi_{\texttt{exit}}}, I'_{\pi_{\texttt{exit}}}) &\overset{\texttt{def}}{=} (\forall X.(I'_{\pi_{\texttt{exit}}} \Rightarrow I_{\pi_{\texttt{exit}}}) \wedge \exists X.(\neg I'_{\pi_{\texttt{exit}}} \wedge I_{\pi_{\texttt{exit}}}))\end{aligned}$$

As before, for maximally weak/strong relations, we cannot directly encode this formula as a SAT instance because of the nested quantification. The situation is additionally complicated because we do not have any assertions (for which we computed the maximally weak preconditions) or any preconditions (for which we computed the maximally strong postconditions) to propagate. In fact, there will be potentially infinite families of summaries that are individually concise, yet incomparable: Intuitively, given a concise summary (A, B), if we use a weaker precondition A' instead of A, then it may be possible to derive another B' that is weaker than B such that (A', B') is a valid concise summary. Note that (A, B) and (A', B') are incomparable.

Example 2.4 *Consider the simple program:*

$$P(x, y)\{\texttt{if}(x \leq y)\ \texttt{then fail};\ \texttt{else return}\ x - y;\}$$

Suppose the template only permits a single linear inequality. Then one concise summary is $(x > y, \texttt{ret} > x - y)$*, but so is* $(x > y + 10, \texttt{ret} > x - y + 10)$*. Notice that the summaries are incomparable.*

Parameterized summaries To express a family of summaries, we discuss the notion of a *parametrized summary.* Notice that the free variables in a standard summary

(A, B) are the formal parameters of the function for A and additionally the return variables for B. In a parametrized summary we also allow a set of free variables that take integral values. Therefore for a function with input x and return value `ret`, instead of a summary $(x > 2, \texttt{ret} > 3)$ we may have a parametrized summary $(x > c, \texttt{ret} > c + 1)$, where c is the additional free variable representing an arbitrary constant. Notice that this allows us to specify an infinite family of summaries, since the new variables are implicitly universally quantified over the domain of integers. Such symbolic summaries have appeared in previous proposals [134, 271] for interprocedural analysis as well.

Parameterized summaries for loop-free programs Parameterized summaries may be trivially obtained for loop-free programs by symbolically executing [165, 120] all paths through a loop free program. Symbolic execution consists of treating the input parameters as symbolic unknowns and then running an interpreter over the program. The interpreter makes calls to a theorem prover when it needs to decide which branch of a conditional to take, and if both branches are feasible given the symbolic constraints then it branches to explore both paths. Summaries generated using symbolic execution may be aggregated by combining pre- and postconditions, if possible. Two summaries can be combined without loss of information, if a new summary can be found that is weaker (respectively, stronger) than both the original summaries in the precondition (respectively, postcondition). In fact, this process can also approximate summary computation for certain well-behaved loops, as has been proposed in the past [12]. Notice that this process will not yield concise summaries by itself, as the input preconditions are constrained to be of the form $\vec{x} = \vec{\alpha}$, i.e., a vector of equalities, where $\vec{\alpha}$ are the initial symbolic values for the formals $\vec{x}$. Thus, the summaries will not be concise unless a complicated semantic merging step is used for postprocessing. For example, for a program $P(x, y)\{\texttt{if}(x > y)\ \texttt{then return}\ 2 * x - y;\ \texttt{else}\ \{\texttt{if}(x = y)\ \texttt{return}\ y;\ \texttt{else fail};\}\}$ symbolic execution will generate two summaries $(x > y, \texttt{ret} = 2x - y)$ and $(x = y, \texttt{ret} = y)$, both of which are not concise as their exists a single concise summary $(x \geq y, 2x-y)$ that is better than them.

Concise parameterized summaries for programs with loops For programs with loops that cannot be approximated using symbolic execution, it may be possible to use an encoding similar to our maximally weak and maximally strong local encodings to generate conciseness constraints.

The key to generating parametrized summaries is to treat the input precondition $I_{\pi_{\texttt{entry}}}$ and output postcondition $I_{\pi_{\texttt{exit}}}$ as unknowns (as before), but to write the output relation's coefficient as a function of the input coefficients. For instance, if $I_{\pi_{\texttt{entry}}}$ is of the form $C_0 + C_1x + C_2y \,.\,. \geq 0$ then the output relation has the form $D_0 + D_1x + D_2y \,.\,. \geq 0$, but where each D_i is a function of the C_i's, i.e., each D_i is $c^i_0 + c^i_1C_1 + c^i_2C_2 + \,.\,. \geq 0$, where c^i_j are the coefficients that the system infers values for. Essentially we are treating the input coefficients C_i's as variables in their own right (thus implicitly universally quantifying them), and the output coefficient D_i's as being a function of the input coefficients.

We then assert that $I_{\pi_{\texttt{entry}}}$ is locally pointwise-weakest and $I_{\pi_{\texttt{exit}}}$ is locally pointwise-strongest. This ensures the (local) conciseness of the summary at the endpoints. Additionally, we need to ensure that the endpoints are consistently connected to each other through intermediate relations for which we assert locally pointwise-weakest/strongest constraints on the intermediate relations. (We conjecture, but have not proven that because of symmetry in this case, that asserting locally pointwise-weakest has the same effect as asserting locally pointwise-strongest. Therefore we can assert either.) Locally pointwise-weakest/strongest constraints ensure that intermediate facts are extremal. Lastly, we iterate to ensure that each summary computed is concise, and additionally once a concise summary is obtained we assert its negation and iterate to compute a summary set that is also precise.

Notice that this encoding will result in quadratic terms, i.e., quadratic in the variables that are universally quantified, which now includes the C_i's, in the resulting formula. We employ a trick of renaming each quadratic term $a * b$ to a new variable a_b to get a constraint system that is linear. This translation is sound but incomplete as it ignores correlations between variables that represent quadratic terms. For example, it may find a constraint system unsatisfiable that relies on implications such as $a = b \Rightarrow a * a = b * b$. While it is incomplete we have found that most programs require little quadratic reasoning, and missing facts can be manually assumed if required, e.g., for the previous example, adding `assume`$(a = b \Rightarrow a\_a = b\_b)$ at appropriate locations would suffice. We discuss this translation more in Chapter 6.2.2.2.

2.7 Applications

In earlier sections, we have described satisfiability-based techniques for verification of safety properties. In this section, we show how to apply those techniques for finding counterexamples to safety properties, verification of termination (an instance of a liveness property), and finding counterexamples to termination.

2.7.1 Termination and Bounds Analysis

The termination problem involves checking whether the given procedure terminates under all inputs. In this section, we show how to use the satisfiability-based approach to solve a harder problem, namely bounds analysis. The problem of bounds analysis is to find a worst-case bound on the running time of a procedure, say in terms of the number of instructions executed, as a function its inputs.

We build on earlier techniques that reduce the bounds analysis problem to discovering invariants of a specific kind [135, 138]. We compute bounds on loop iterations and the number of recursive procedure call invocations. Each of these can be bounded by appropriately instrumenting counter variables and estimating bounds on counter variables. We instrument loops "while c do S" by adding a counter i to get "$i := 0$; while c do $\{$ i++; S; $\}$". The number of loop iterations are then bounded by computing an upper bound on the value of i. We instrument recursive procedures "$P(x)$ $\{$ S $\}$" by adding a counter i to get "$P(x)$ $\{$ $i := 0; P'(x)$; $\}$; $P'(x')$ $\{$ i++; $S[x'/x]$; $\}$". the number of invocations of the procedure are then bounded by computing an upper bound of the value of the global variable i.

Claim 2.1 *Let P be a given program. Let P' be the transformed program obtained after instrumenting counters that keep track of loop iterations and recursive call invocations and introducing partial assertions that the counters are bounded above by some function of the inputs. The program P terminates iff the assert statements in P' are satisfied.*

Invariant generation tools based on abstract interpretation have been proposed for computing bounds on the counter variables [138, 135]. We show instead that a satisfiability-based approach is particularly suited for discovering these invariants since they have a specified form and involve linear arithmetic. We introduce assert statements with templates $i < \sum_k a_k x_k$ (at the instrumented site i++ for loops and at the end of the procedure for recursive procedures) for bounding the counter value. Observe that the bounds templates that we have introduced are linear. Instrumentation can be used to compute non-linear bounds as a composition of linear bounds on multiple counters [138, 135].

Additionally, the satisfiability-based approach solves an even harder problem, namely inferring preconditions under which the procedure terminates and inferring a bound under that precondition. For this, we introduce the bound templates on instrumented counter variables as described above and infer maximally weak preconditions. This is significant for procedures that only terminate under certain preconditions and not for all inputs. We are not aware of any other technique that can compute such conditional bounds.

```
Loop(int $n$, $m$) {
  $x := x_0$; $y := y_0$;
  while ($x < y$)
      $x := x + n$;
      $y := y + m$;
}
```

Original Program

```
Loop(int $n$, $m$) {
  $x := x_0$; $y := y_0$; $i := 0$;
  while ($x < y$)
     $i$++;
     $x := x + n$;
     $y := y + m$;
}
```

Instrumented Program

Figure 2.9: Discovering maximally weak preconditions for termination.

```
Fib(int $n$) {
  if($n = 0$)
    return 1;
  else
    return Fib($n - 1$);
}
```

Original Program

```
Fib(int $n$) {
  $i := 0$
  return Fib′($n$)
}
Fib′(int $n'$) {
  $i$++;
  if($n' = 0$)
    return 1;
  else
    return Fib′($n' - 1$);
}
```

Instrumented Program

Figure 2.10: Termination in the presence of recursion

Example In Figure 2.9 we compute three relations: the maximally weak precondition at the beginning of the procedure, the bound on the instrumentation counter at the counter increment site, and the loop invariant at the header. Our tool computes the precondition $n \geq m + 1$ and the bound $y_0 - x_0$. The latter requires discovering the inductive loop invariant $i < (x - x_0) - (y - y_0)$.

Example Consider the recursive procedure shown in Figure 2.10. The instrumentation introduces an auxiliary function Fib′, and we compute three relations: the maximally weak precondition at the beginning of Fib, the procedure summary for Fib′, and the invariant $i < a_0 + a_1 n$ at the counter instrumentation point. Our tool computes the precondition $n \geq 0$ at the entry to Fib(n), and the bound $i \leq n$ inside Fib. The latter requires discovering the summary pair $(n' > 0, i^{out} - i^{in} \leq n')$. This example illustrates interprocedural maximally weak precondition inference.

2.7.2 Counterexamples for Safety Properties

Since program analysis is an undecidable problem, tools cannot prove the correctness of arbitrary correct programs or find bugs in arbitrary incorrect programs. Hence, to maximize the practical success rate of verification tools, it is desirable to search in parallel for both proofs of correctness as well as counterexamples. Earlier, we showed how to find proofs of correctness of safety and termination properties. In this section, we show how to find *most-general* counterexamples to safety properties. A safety property is stated as set of *safety assertions*. A violation of the safety property occurs if the negation of a safety assertion holds and is reachable.

The problem of most general counterexample for safety involves finding the most general characterization of inputs that leads to the violation of some reachable safety assertion. We show how to find such a characterization using the techniques discussed in Section 2.4 and Section 2.7.1.

The basic idea is to reduce the problem to that of finding the maximally weak precondition for some safety property. This reduction involves constructing another program from the given program P using the following transformations:

B1 *Instrumentation of program with an error variable* We introduce a new error variable that is set to 0 at the beginning of the program. Whenever violation of the given safety property occurs (the negation of the safety assertions holds), we set the error variable to 1 and jump to the end of the program, where we assert that the error variable is equal to 1. We remove the original safety assertion.

B2 *Instrumentation to ensure termination of all loops* For this we use the strategy described in Section 2.7.1, wherein we instrument the program with counter variables and assert that the counter variable is upper bounded by some function of loop inputs or procedure inputs. The function is modeled using a linear arithmetic template for which we infer the coefficients.

Claim 2.2 *Let P be a program with some safety assertions. Let P' be the program obtained from program P by using the transformation B1 and B2 above. Then, P has an assertion violation iff the assertions in program P' hold.*

Claim 2.2 is significant as we can now use maximally weak precondition inference (Section 2.4) on the transformed program to discover most-general characterization of inputs under which there is a safety violation in the original program.

Example The program shown in Figure 2.11(a) is instrumented using transforms B1 and B2, and the resulting program is shown in Figure 2.11(b). Our tool discovers the precondition $(n > 200) \wedge (9 > y > 0)$. The loop invariant that asserts termination of the relevant loop on line 3 is $(n > 200) \wedge (i \leq x) \wedge (9 > y > 0) \wedge (x \leq 200)$. A loop bound using the function $i < n + 1$ proves that the loop terminates. On the other

```
  Bug1(int $y$, $n$) {
1   $x := 0$;
2   if($y < 9$)
3     while ($x < n$)
4         assert($x < 200$);
5         $x := x + y$;
6   else
7     while ($x \geq 0$)
8         $x$++;
  }
```

Original Program

```
   Bug1(int $y$, $n$) {
1    $x := err := i_1 := i_2 := 0$;
2    if($y < 9$)
3      while ($x < n$)
4          $i_1$++;
5          assert($i_1 < f_1(n, y)$);
6          if($x \geq 200$)
7            $err := 1$; goto $L$;
8          $x := x + y$;
9    else
10     while ($x \geq 0$)
11         $i_2$++;
12         assert($i_2 < f_2(n, y)$);
13         $x$++;
14 $L$:  assert($err = 1$);
   }
```

Instrumented Program

Figure 2.11: The most general counterexample that leads to violation of the safety assertion in the original program is $(n > 200) \wedge (0 < y < 9)$. Our tool discovers this by instrumenting the program appropriately and then running our maximally weak precondition algorithm.

hand, since the loop on line 10 is unreachable under the discovered preconditions an arbitrary f_2 suffices.

Observe the importance of transformation B1. An alternative to transformation B1 that one might consider is to simply negate the original safety assertion instead of introducing an error variable. This is incorrect for two reasons: (a) It is too stringent a criterion because it insists that in each iteration of the loop the original assertion does not hold, and (b) It does not ensure reachability and allows for those preconditions under which the assert statement is never executed at all. In fact, when we run our tool with such a naive transformation that simply negates the safety assertion, we obtain $n \leq 0$ as the maximally weak precondition.

Also, observe the importance of transformation B2. If we do not perform

```
NT1(int x, y) {
  while (x ≥ 0)
     x := x + y;
     y++;
}
```

(a)

```
NT2(int i) {
  even := 0;
  while (i ≥ 0)
     if (even = 0)
        i--;
     else
        i++;
     even := 1 − even;
}
```

(b)

Figure 2.12: Non-termination examples from an alternative approach [142].

transformation B2, then the tool discovers $y \leq 0$ as the maximally weak precondition. Note that under this precondition, the assertion at the end of the program always holds since that location is unreachable. Observe that the transformation B2 does not require termination of every loop in the original program. In fact, violation of safety properties can also occur in non-terminating programs. The transformation B2 ensures termination of all loops that are reachable *under the precondition that the tool discovers* and in the program obtained after transformation B1, which introduces extra control-flow that breaks loops on any violation of a safety property. This is the case for the loop on line 10, which is unreachable under the discovered preconditions and therefore any arbitrary function f_2 suffices.

2.7.3 Counterexamples for Termination Properties

The problem of inferring most-general counterexamples for termination properties involves finding the most-general characterization of inputs that leads to non-termination of the program. Without loss of generality we assume that the program has at most one exit point.

Claim 2.3 *Let P be a given program with a single exit point. Let P' be the program obtained from P by adding the assert statement "*`assert`*$(false)$" at the end of the program. Then, P is non-terminating iff the assert statement in P' is satisfied.*

By Claim 2.3, we can use maximally weak precondition inference (Section 2.4) on the transformed program to discover preconditions for non-termination.

Examples Consider the example shown in Figure 2.12(a). If we instrument the program to add a `assert`$(false)$ at the end, then our maximally weak precondition algorithm generates the constraint $x \geq 0 \wedge y \geq 0$, which is the maximally weak condition under which the program is non-terminating.

Consider program shown in Figure 2.12(b). If we instrument `assert`$(false)$ at the end of this program, then our maximally weak precondition inference generates the condition $i \geq 1$. Notice that the loop guard $i \geq 0$ is not sufficient to guarantee non-termination. A recent proposal [142] for proving non-termination searches for *recurrent* sets of states and will have to unroll the loop to reason about the value of `even`. We never unroll loops and additionally discover the maximally weak preconditions that ensure non-termination.

2.8 Experiments

In previous sections, we have shown how to model various program analysis problems as the problem of solving SAT constraints. We now present encouraging experimental results illustrating that SAT solvers can in fact efficiently solve the

SAT instances generated using our technique. Our examples come directly from benchmarks used in state-of-the-art alternative techniques. We employ an incremental strategy for choosing the template. We progressively increased the number of bits in the bit-vector modeling and the number conjuncts and disjuncts if the SAT solver proves the initial instances UNSAT, until the solver found a SAT solution, and thus inferred the invariants. In practice, we never had to go beyond two iterations. In Tables 2.1, 2.2, 2.3, and 2.4 we present the programs, the time taken in seconds for constraint generation and constraint solving, and the number of clauses in the CNF formula. We provide sources and/or figure references from previous sections for most examples and explain the remainder.

We ran the experiments on a two processor machine running Windows Vista$^{\text{TM}}$ and used Z3 [86] as our SAT/SMT solver. We experimented with various other solvers (ZChaff [203] and its variants, Minisat [99] etc) but found Z3 to be the most efficient at solving the constraints generated for the benchmark programs. We have noticed that symmetry in the satisfiability problem, seen for instance in the case of discovering disjunctive invariants, causes significant degradation of performance. The solver could potentially use the symmetry information to prune its search space. In future work, we expect to modify the solver to use this higher level domain information. More details about engineering a satisfiability-based invariant generation tool are presented in Chapter 6.

Even with our unoptimized prototype implementation the constraint generation phase takes from between $0.09 - 0.30$ seconds across all benchmarks. This includes the overhead of reading and parsing the program from disk and CFG gen-

Name	**Constraint Gen. Time (s)**	**Solving Time (s)**	**Number Clauses**
cegar1 [133]	0.09	0.08	5 K
cegar2 [133]	0.10	0.80	50 K
barbr [132]	0.15	0.41	76 K
berkeley [132]	0.13	3.00	441 K
bk-nat [132]	0.15	5.30	174 K
ex1 [132]	0.10	0.10	10 K
ex2 [132]	0.10	0.75	92 K
fig1a [132]	0.10	0.14	20 K
fig2 [132]	0.10	0.56	239 K
fig3 [132]	0.14	16.60	547 K
w1 [34], pg12	0.10	0.14	25 K
w2 [34], pg12	0.10	1.80	165 K

Table 2.1: Program verification over linear arithmetic.

eration and the time to write the constraints to disk. Many of these phases can be optimized—e.g., by eliminating writing intermediate phases to disk—but we leave that to future work. This illustrates the scalability of our reductions. The constraint solving phase is listed separately because it depends on the particular solver being used and its current version, Z3 v1.0 for our case. The total time for constraint solving varies from 0.08 to 72.00 seconds. Improvements in solver technology will directly translate to decrease in these numbers.

Table 2.1 presents program verification analysis on examples taken from abstraction refinement-based techniques [133, 132] and programs for which standard widening/narrowing fails [34]. We ran our tool on benchmarks considered in state-of-the-art alternative verification techniques [133, 132] because they provide exhaustive comparison against techniques similar to theirs. `w1` is a simple loop iteration but with $x \leq n$ replaced with $x \neq n$ while `w2` is a loop with the guard moved inside a non-deterministic conditional. Standard narrowing is unable to capture the preci-

Name	**Constraint Gen. Time (s)**	**Solving Time (s)**	**Number Clauses**
Fig 2.3(a), [238]	0.09	0.57	63 K
a1 [206], pg9	0.11	9.90	174 K
a2 [204], pg2	0.15	0.50	75 K
mergesort	0.09	0.19	43 K
quicksort	0.09	0.45	133 K
fibonacci	0.10	11.00	90 K
Fig 2.3(b)	0.20	72.00	558 K

Table 2.2: Interprocedural analysis over linear arithmetic.

Name	**Constraint Gen. Time (s)**	**Solving Time(s)**	**Number Clauses**
Fig 2.2 [125, 126]	0.20	0.70×2	107 K
Fig 2.8	0.20	5.70×3	273 K
w1 [34], pg 12	0.10	0.30×2	60 K
burner [124], pg 14	0.20	1.50×1	100 K
speed [126], pg 10	0.20	9.10×2	41 K
merge [125], pg 11	0.20	1.30×3	128 K

Table 2.3: Maximally strong postcondition inference over linear arithmetic.

sion lost due to widening in these instances. Our solution times compare favorably against previous techniques.

Table 2.2 presents interprocedural analysis results on benchmarks from alternate proposals [204, 206, 238]. The first benchmark is the recursive add from Figure 2.3(a). The second `a1` and third `a2` programs rely on discover linear equality relations for recursive procedures. The fourth and fifth are recursive sorting programs and the sixth is the Fibonacci program. The last benchmark in the set is the McCarthy91 function from Figure 2.3(b), for which we compute two summaries.

Table 2.3 presents maximally strong postconditions generation results on benchmarks from papers on sophisticated widening techniques [34, 124, 125, 126]. For our iterative algorithm we present the times taken for each iteration and the number of

Name	**Constraint Gen. Time (s)**	**Solving Time (s)**	**Number Clauses**
[142], pg3	0.15	0.80×1	42 K
Fig 2.12(b) [142], pg5	0.19	0.40×1	57 K
Fig 2.12(a) [142], pg5	0.16	0.60×1	43 K
loop	0.14	0.12×1	15 K
Fig 2.5(a)	0.18	3.80×4	119 K
Fig 2.5(b)	0.27	40.00×2	221 K
Fig 2.7	0.23	0.50×1	50 K
Fig 2.9	0.15	11.60×1	118 K
Fig 2.11	0.30	34.00×2	135 K

Table 2.4: Weakest precondition inference over linear arithmetic (including non-termination and bug-finding examples).

iterations in the timings column. This provides finer insight into the time taken for generating each maximally strong postcondition, as opposed to just the total. `w1`, `burner`, `speed` and `merge` model hybrid automaton for real systems and even our prototype timings are encouraging, so we are confident that our technique will be practical.

For maximally weak precondition generation (as in maximally strong postcondition) we present, as before, the time for each iteration times the number of iterations. The first set in Table 2.4 presents results on analysis of non-termination programs `nt1/nt2/nt3` [142] and shown in Figures 2.12(a) and 2.12(b). Our technique also facilitates maximally weak precondition generation for examples such as array increment and array copy and swap (Figures 2.5(a), 2.5(b) and 2.8(a)) which our tool analyzes in reasonable time. We also find the maximally weak preconditions for termination for Figure 2.9. Lastly, generating maximally weak precondition for our most intriguing example (Figure 2.11) takes 68 seconds.

2.9 Summary

This chapter described how to model a wide spectrum of program analysis problems as SAT instances that can be solved using off-the-shelf constraint (SAT) solvers. We showed how to model the problem of discovering invariants, both conjunctive and disjunctive, that involve linear inequalities. We applied it to intra- and interprocedural checking of safety properties and timing analysis of programs. We also showed how to model the problem of discovering maximally weak preconditions and maximally strong postconditions. We applied pre- and postcondition inference towards generating most-general counterexamples for both safety and termination properties.

The constraints that we generate are boolean combinations of quadratic inequalities over integer variables, which we reduce to SAT formulas using bit-vector modeling. We showed experimentally that the SAT solver can efficiently solve such constraints generated from hard benchmarks.

2.10 Further Reading

Contrast with tradition It is important to compare the benefits and limitations of a satisfiability-based approach against traditional iterative fixed-point approximation techniques, such as data-flow analyses, abstract interpretation and model checking.

The key difference between a satisfiability-based approach and traditional techniques is the lack of iterative approximations. By encoding the problem as a solution to a SAT instance, we are able to delegate fixed-point solving to the SAT solver, and verification is non-iterative. Only when we deal with the more sophisticated problem of weakest precondition/strongest postcondition inference do we have to resort to iteration, and that too only when enumerating orthogonal solutions, or when dealing with programs with local minimas.

Additionally, we note two advantages of a satisfiability-based approach. First, a satisfiability-based approach is goal-directed and hence has the potential to be more efficient. The data-flow analyses or abstract interpreters typically work either in a forward direction or in a backward direction, and hence are not goal-directed. Some efforts to incorporate goal-directedness involve repeatedly performing a forward (or backward) analysis over refined abstractions obtained using counterexample guidance, or by repeatedly iterating between forward and backward analyses [77]. However, each forward or backward analysis attempts to compute the most precise information over the underlying domain, disregarding what might really be needed. On the other hand, the satisfiability-based approach is fully goal-directed; it abstracts away the control-flow of the program and incorporates information from both the precondition as well as the postcondition in the constraints. Second, a satisfiability-based approach does not require widening heuristics, that can lead to uncontrolled loss of precision, but are required for termination of iterative fixed-point techniques. Abstract interpreters iteratively compute approximations to fixed-points and use domain-specific extrapolation operators (widening) when operating over infinite height lattices (e.g., lattice of linear inequalities) to ensure termination. Use of widening leads to an uncontrolled loss of precision. This has led to development of several widening heuristics that are tailored to specific classes of programs [264, 132, 125, 126]. We show that the satisfiability-based approach can uniformly discover invariants for all such programs.

We now note some disadvantages of a satisfiability-based approach. First, the execution time of analyses in this framework is less deterministic as it is dependent on the efficiency of the underlying SAT solver. In preliminary tests, we found competitive efficiency but only further experiments will demonstrate the true limitations of this approach. Second, a domain-specific technique, namely Farkas' Lemma, enabled the reduction of program constraints to satisfiability constraints. In the next chapter, we will see an algorithm for a predicate abstraction of programs that reduces the problem to satisfiability constraints. Such domain specific reductions are necessarily required for our approach and for earlier ones (e.g., join, widen, and transfer functions in abstract interpretation). The key to successfully exploiting the power of a satisfiability-based framework for program analysis will be the development of novel domain specific reductions.

Using satisfiability-based approaches Ideas similar to the ones presented here, have been explored by others in developing efficient program analysis solutions. InvGen generates SAT instances that are simpler to solve by augmenting the core constraints with constraints over a set of symbolic paths (e.g., from tests) [143, 144]. Constraint-based solutions find applications in hardware synthesis [68]. For inferring dependent types, specifically, ML types refined by linear relations, liquid types [229, 161] generates and solves constraints over the refinements, and can benefit from a satisfiability-based approach.

Chapter 3

Program Reasoning over Predicate Abstraction

"Besides black art, there is only automation and mechanization."

— Federico Garcia Lorca[1]

In this chapter, we augment the expressivity of the invariant generation approach of the previous chapter by inferring invariants over predicate abstraction. We describe how a satisfiability-based approach over predicate abstraction can discover invariants with quantified and arbitrary boolean structure. These then help us prove the validity of given assertions or generating pre-conditions under which the assertions are valid. We present three novel algorithms, having different strengths, that combine template-and predicate abstraction-based formalisms to discover sophisticated program invariants using SMT solvers.

Two of these algorithms use an iterative approach to compute least and greatest fixed-points, while the third algorithm uses a non-iterative satisfiability-based approach that is similar in spirit to the approach for linear arithmetic. The key idea for predicate abstraction in all these algorithms is to reduce the problem of invariant discovery to that of finding *optimal* solutions, over conjunctions of some predicates

[1] Spanish poet, dramatist and theater director, 1898-1936.

from a given set, for unknowns in a template formula.

We have implemented the algorithms presented in this chapter in a tool that we call $\mathtt{VS}^3_{\mathtt{PA}}$. Preliminary experiments using $\mathtt{VS}^3_{\mathtt{PA}}$ show encouraging results over a benchmark of small but complicated programs. Our algorithms can verify program properties that, to our knowledge, have not been automatically verified before. In particular, our algorithms can generate full correctness proofs for sorting algorithms by inferring nested universally-existentially quantified invariants, and can also generate preconditions required to establish worst-case upper bounds of sorting algorithms. Furthermore, for properties that can be verified by previous approaches, our tool is more efficient.

3.1 Using SMT Solvers for Program Reasoning

In this chapter, we continue our discussion on template-based program analysis that shows promise in discovering invariants that are beyond the reach of fully automated techniques. The programmer provides hints in the form of a set of invariant templates with holes/unknowns that are then automatically filled in by the analysis. However, in the previous chapter we discussed quantifier-free numerical invariants, also considered in previous work [233, 234, 62, 160, 28, 137]). In contrast, in this chapter we consider invariants with arbitrary but pre-specified logical structure—involving disjunctions and universal and existential quantifiers—over a given set of predicates. One of the key features of our template-based approach is that it uses the standard interface to an SMT solver, allowing it to go beyond numerical properties and leverage ongoing advances in SMT solving.

Our templates consist of formulae with arbitrary logical structure (quantifiers, boolean connectives) and unknowns that take values over some conjunction of a given set of predicates (Section 3.3). Such a choice of templates puts our work in an unexplored space in the area of predicate abstraction, which has been highly successful in expressing useful non-numerical and disjunctive properties of programs. The area was pioneered by Graf and Seidl [128], who showed how to compute quantifier-free invariants over a given set of predicates. Later, strategies were proposed to discover universally quantified invariants [112, 176, 154] and disjunctions of universally quantified invariants in the context of shape analysis [220]. Our work extends the field by discovering invariants that involve an arbitrary (but pre-specified quantified structure) over a given set of predicates. Since the domain is finite, one can potentially search over all possible solutions, but this naive approach would be too computationally expensive to be feasible.

We therefore present three novel algorithms for efficiently discovering inductive loop invariants that prove the validity of assertions in a program, given a suitable set of invariant templates and a set of predicates. Two of these algorithms use iterative techniques, unlike the SAT-based approach presented in the previous chapter, for computing fixed-point as in data-flow analysis or abstract interpretation. One of them performs a forward propagation of facts and computes a least fixed-point, and then checks whether the facts discovered imply the assertion or not (Section 3.5.1). The other algorithm performs a backward propagation of facts starting from the given assertion and checks whether the precondition discovered is *true* or not (Section 3.5.2). The third algorithm uses a satisfiability-based approach, akin to the approach in the previous chapter, to encode the fixed-point as a SAT formula such that a satisfying assignment to the SAT formula maps back to a proof of validity for the assertion (Section 3.6). The worst-case complexity of these algorithms is exponential only in the maximum number of unknowns at two neighboring points as opposed to being exponential in the total number of unknowns at all program points for the naive approach. Additionally, in practice we have found them to be efficient and having different strengths (Section 3.8).

The key operation in these algorithms is that of finding *optimal solutions* for unknowns in a template formula such that the formula is valid (Section 3.4). The unknowns take values that are conjunctions of some predicates from a given set of predicates, and can be classified as either positive or negative depending on whether replacing them by a stronger or weaker set of predicates makes the formula stronger or weaker respectively. We describe an efficient, systematic, search process for finding optimal solutions to these unknowns. Our search process uses the observation that a solution for a positive (or negative) unknown remains a solution upon addition (or deletion) of more predicates.

One of the key aspects of our algorithms is that they can be easily extended to discover *maximally weak* preconditions. This is unlike most invariant generation tools that cannot be easily extended to generate pre-conditions, especially those that are maximally weak. Automatic precondition generation not only reduces the annotation burden on the programmer in the usual case, but can also help identify preconditions that are not otherwise intuitive.

3.2 Motivating Examples

Inferring invariants for checking assertions Consider the in-place `InsertionSort` routine in Figure 3.1 that sorts an array A of length n. The assertion at Line 9 asserts that no elements in array A are lost, i.e., the array A at the end of the procedure contains all elements from array $\tilde{A}$, where $\tilde{A}$ refers to the state of array A at the beginning of the procedure. The assertion as well as the loop invariants required to prove it are $\forall\exists$ quantified, and we do not know of any other automated tool that can automatically discover such invariants for array programs.

In this case, the user can easily guess that the loop invariants would require a $\forall\exists$ structure to prove the assertion on Line 9. Additionally, the user needs to guess that an inductive loop invariant may require a $\forall$ fact (to capture properties of array elements) and a quantifier-free fact relating non-array variables. The quantified facts contain an implication as in the final assertion. The user also needs to provide the set of predicates. In this case, the set consisting of inequality and disequality comparisons between terms (variables and array elements that are indexed by some variable) of appropriate types suffices. This choice of predicates has been used successfully in previous work on predicate abstraction [15, 11, 175, 176]. Given these user inputs, our tool then automatically discovers the non-trivial loop invariants mentioned in the figure.

As a second example, consider the program shown in Fig. 3.2, which checks whether all elements of A are contained in B. The loop invariant required contains $\forall\exists$ quantification, which our tool can infer. We do not know of any other tool that can automatically discover such invariants. Note how the conjuncts in the invariant template in this case follow the schematic of the given assertion and therefore are $\forall\exists$-quantified. We discovered the appropriate number of conjuncts by iteratively guessing templates.

Our tool eases the task of validating the assertion by requiring the user to only provide a template in which the logical structure has been made explicit, and provide some over-approximation of the set of predicates. Guessing the template is a much easier task than providing the precise loop invariants, primarily because these templates are usually uniform across the program and depend on the kind of properties to be proved.

Precondition Generation Consider the in-place `SelectionSort` routine in Figure 3.3. This routine sorts an array A of length n. Suppose we want to verify that the worst-case number of array swaps is indeed $n-1$. This problem can be reduced to the problem of validating the assertion at Line 7. If the assertion holds then the swap on Line 8 is always executed, $n-1$ times [135]. However, this assertion is not valid without an appropriate precondition, e.g., consider a fully sorted array for which no swaps happen. We want to find a precondition that does not impose any constraints on n while allowing the assertion to be valid. This would provide a proof that `SelectionSort` indeed admits a worst-case of $n-1$ memory writes.

In this case, the user can easily guess that a quantified fact—$\forall k_1, k_2$ that compares the elements at locations k_1 and k_2—will capture the sortedness property that is required. However, this alone does not yield the correct invariants. The user

```
InsertionSort(Array $A$, int $n$)
  $i := 1$;
  while $(i < n)$
      $j := i - 1$; val $:= A[i]$;
      while $(j \geq 0 \wedge A[j] >$ val$)$
          $A[j+1] := A[j]$;
          $j := j - 1$;
      $A[j+1] :=$ val;
      $i := i + 1$;
  Assert$(\forall y \exists x : (0 \leq y < n) \Rightarrow (\tilde{A}[y] = A[x] \wedge 0 \leq x < n))$
```

User Input:

Invariant Template: $v_1 \ \wedge \ (\forall y : v_2 \Rightarrow v_3) \ \wedge \ (\forall y \exists x : v_4 \Rightarrow v_5)$

Predicate Set: $\texttt{AllPreds}(\{x, y, i, j, n\}, \{0, \pm 1\}, \{\leq, \geq, \neq\}) \ \cup$
$\texttt{AllPreds}(\{\texttt{val}, A[t], \tilde{A}[t] \mid t \in \{i, j, x, y, n\}\}, \{0\}, \{=\})$

Tool Output:

(Proof of validity of assertion)

Outer Loop Invariant: $\begin{pmatrix} \forall y : (i \leq y < n) \Rightarrow (\tilde{A}[y] = A[y]) \ \wedge \\ \forall y \exists x : (0 \leq y < i) \Rightarrow (\tilde{A}[y] = A[x] \wedge 0 \leq x < i) \end{pmatrix}$

Inner Loop Invariant: $\begin{pmatrix} \texttt{val} = \tilde{A}[i] \wedge -1 \leq j < i \ \wedge \\ \forall y : (i < y < n) \Rightarrow \tilde{A}[y] = A[y] \ \wedge \\ \forall y \exists x : (0 \leq y < i) \\ \qquad \Rightarrow (\tilde{A}[y] = A[x] \wedge 0 \leq x \leq i \wedge x \neq j + 1) \end{pmatrix}$

Figure 3.1: Verifying that insertion sort preserves all its input elements $\texttt{AllPreds}(Z, C, R)$ denotes the set of predicates $\{z - z' \ \texttt{op} \ c, \quad z \ \texttt{op} \ c \mid z, z' \in Z, c \in C, \texttt{op} \in R\}$.

```
SetInclusion(Array $A$, int $n$, Array $B$, int $m$)
for ($i = 0; i < n; i$++)
    $exists$ := $false$;
    for ($j = 0; j < m; j$++)
        if ($A[i] = B[j]$)
            $exists$ := $true$; break;
    if ($\neg exists$) return false;
Assert ($\forall y \exists x : (0 \leq y < n)$
            $\Rightarrow (A[y] = B[x] \wedge 0 \leq x < m)$)
return true;
```

User Input:

Invariant Template: $v_1 \ \wedge \ (\forall y \exists x : v_2 \Rightarrow v_3) \ \wedge \ (\forall y \exists x : v_4 \Rightarrow v_5)$

Predicate Set: $\texttt{AllPreds}'(\{x, y, i, j, m\}, \{0\}, \{\leq, <\}) \ \cup$
$\texttt{AllPreds}'(\{exists\}, \{true, false\}, \{=\}) \ \cup$
$\texttt{AllPreds}'(\{A[t], B[t] \mid t \in \{x, y\}\}, \{0\}, \{=\})$

Tool Output:

(Proof of validity of assertion)

Outer loop invariant: $\big(\ \forall y \exists x : (0 \leq y < i) \Rightarrow (A[y] = B[x] \wedge 0 \leq x < m) \ \big)$

Inner loop invariant:
$$\left(\begin{array}{l} j \geq 0 \\ \forall y \exists x : (0 \leq y < i) \Rightarrow (A[y] = B[x] \wedge 0 \leq x < m) \\ \forall y \exists x : (y = i \wedge exists = true) \\ \qquad\qquad\qquad \Rightarrow (A[y] = B[x] \wedge 0 \leq x < m) \end{array} \right)$$

Figure 3.2: Verifying that a program that checks set inclusion is functionally correct. VS^3 computes the $\forall\exists$ invariants required to prove the correctness. $\texttt{AllPreds}'(Z, C, R)$ denotes the set of predicates $\{z \ \texttt{op} \ z' \mid z, z' \in Z \cup C, \texttt{op} \in R\}$.

then iteratively guesses and adds templates until a precondition is discovered. Two additional quantified facts and an unquantified fact suffice in this case. While right now this process is manual, in the future it we can expect it can be automated. The user also supplies a predicate set consisting of inequalities and disequalities between terms of comparable types. The non-trivial output of our tool is shown in the figure.

Our tool automatically infers the maximally weak precondition that the input array should be sorted from $A[0]$ to $A[n-2]$, while the last entry $A[n-1]$ contains the smallest element. Other sorting programs usually exhibit their worst-case behaviors when the array is reverse-sorted. For selection sort, a reverse sorted array is not the worst case; it incurs only $\frac{n}{2}$ swaps. By automatically generating this maximally weak precondition our tool provides significant insight about the algorithm, reducing programmer burden.

As another example, consider the program shown in Fig. 3.4, which implements a binary search for the element e in an array A. The functional specification of the program is given as the assertion on Line 9, which states that if the procedure returns false, then A indeed does not contain e. Our tool allows the user to specify assertions and assumptions with arbitrary logical structure up to those expressible in the underlying SMT solver. Assumptions may be required to model expressions not handled by the solver. For instance, since SMT solvers currently do not handle division, the assignment on Line 3 is modeled as `Assume`$(low \leq mid \leq high)$.

For this function, our tool automatically infers the maximally weak precondition for functional correctness, shown in Fig. 3.4, which is that the input array is sorted. It also infers the loop invariant, also shown in Fig. 3.4, encoding the seman-

```
SelectionSort(int* $A$, int $n$)
 $i := 0$;
 while $(i < n - 1)$
     min $:= i$;  $j := i + 1$;
     while $(j < n)$
         if $(A[j] < A[\texttt{min}])$ min $:= j$;
         $j := j + 1$;
     Assert$(i \neq \texttt{min})$;
     if $(i \neq \texttt{min})$ swap $A[i]$ and $A[\texttt{min}]$;
     $i := i + 1$;
```

User Input:

Template: $\big(\ v_0 \wedge (\forall k : v_1 \Rightarrow v_2) \wedge (\forall k : v_3 \Rightarrow v_4) \wedge (\forall k_1, k_2 : v_5 \Rightarrow v_6)\ \big)$

Predicate Set: $\left(\begin{array}{l} \texttt{AllPreds}(\{k, k_1, k_2, i, j, \texttt{min}, n\}, \{0, 1\}, \{\leq, \geq, >\})\ \cup \\ \texttt{AllPreds}(\{A[t] \mid t \in \{k, k_1, k_2, i, j, \texttt{min}, n\}\}, \{0, 1\}, \{\leq, \geq\}) \end{array}\right)$

Tool Output:

(Assertion valid under following precondition)

Precondition Required: $\left(\begin{array}{l} \forall k : (0 \leq k < n - 1) \Rightarrow A[n-1] < A[k] \\ \forall k_1, k_2 : (0 \leq k_1 < k_2 < n - 1) \Rightarrow A[k_1] < A[k_2] \end{array}\right)$

Outer Loop Invariant: $\left(\begin{array}{l} \forall k_1, k_2 : (i \leq k_1 < k_2 < n - 1) \Rightarrow A[k_1] < A[k_2] \\ \forall k : i \leq k < n - 1 \Rightarrow A[n-1] < A[k] \end{array}\right)$

Inner Loop Invariant: $\left(\begin{array}{l} \forall k_1, k_2 : (i \leq k_1 < k_2 < n - 1) \Rightarrow A[k_1] < A[k_2] \\ \forall k : (i \leq k < n - 1) \Rightarrow A[n-1] < A[k] \\ \forall k : (i \leq k < j) \Rightarrow A[\texttt{min}] \leq A[k] \\ j > i \wedge i < n - 1 \end{array}\right)$

Figure 3.3: Generating the weakest precondition under which Selection Sort exhibits its worst-case number of swaps.

tics of binary search (that the array elements between *low* and *high* are sorted and those outside do not equal e).

In the following sections, we develop the theory over predicate abstraction that helps us build tools that can analyze and infer the expressive properties illustrated here.

```
BinarySearch(Array $A$, int $e$, int $n$)
  $low := 0; high := n - 1$;
  while $(low \leq high)$
      $mid := \lceil (low + high)/2 \rceil$;
      if $(A[mid] < e)$
          $low := mid + 1$;
      else if $(A[mid] > e)$
          $high := mid - 1$;
      else return true;
  Assert $(\forall j : (0 \leq j < n) \Rightarrow A[j] \neq e)$
  return false;
```

User Input:

Invariant Template: $v_1 \wedge (\forall j : v_2 \Rightarrow v_3) \wedge (\forall j : v_4 \Rightarrow v_5) \wedge (\forall j : v_6 \Rightarrow v_7)$

Predicate Set: $\texttt{AllPreds}'(\{j, n, low, high\}, \{0\}, \{\leq, <\}) \cup$
$\texttt{AllPreds}'(\{A[t] \mid t \in \{j, j \pm 1\}\} \cup \{e\}, \{0\}, \{\leq, \neq\})$

Tool Output:

(Assertion valid under the following precondition)

Precondition: $\left(\forall j : (0 \leq j < n) \Rightarrow A[j] \leq A[j+1] \right)$

Loop Invariant:
$$\left(\begin{array}{l} 0 \leq low \wedge high < n \\ \forall j : (low \leq j \leq high) \Rightarrow A[j] \leq A[j+1] \\ \forall j : (0 \leq j < low) \Rightarrow A[j] \neq e \\ \forall j : (high < j < n) \Rightarrow A[j] \neq e \end{array} \right)$$

Figure 3.4: Generating the weakest precondition for the functional correctness of binary search.

3.3 Notation

We often use a set of predicates in place of a formula to mean the conjunction of the predicates in the set. In our examples, we often use predicates that are inequalities between a given set of variables or constants. We use the notation Q_V to denote the set of predicates $\{v_1 \leq v_2 \mid v_1, v_2 \in V\}$. We use the notation $Q_{j,V}$ to denote the set of predicates $\{j < v, j \leq v, j > v, j \geq v \mid v \in V\}$. Also, we will use the notation $\{x_i\}_i$ as an abbreviation to a set of indexed variables $\{x_i \mid x_i \in X\}$, if the domain/universe X of the elements x_i's is explicit from their type.

3.3.1 Templates for Predicate Abstraction

A *template* τ is a formula over unknown variables v_i that take values over (conjunctions of predicates in) some subset of a given set of predicates. We consider the following language of templates:

$$\tau \quad ::= \quad v \quad | \quad \neg\tau \quad | \quad \tau_1 \vee \tau_2 \quad | \quad \tau_1 \wedge \tau_2 \quad | \quad \exists x : \tau \quad | \quad \forall x : \tau$$

We denote the set of unknown variables in a template τ by $\texttt{Unk}(\tau)$. We say that an unknown $v \in \texttt{Unk}(\tau)$ in template τ is a *positive (or negative) unknown* if τ is monotonically stronger (or weaker respectively) in v. More formally, let v be some unknown variable in $\texttt{Unk}(\tau)$. Let σ_v be any substitution that maps all unknown variables v' in $\texttt{Unk}(\tau)$ that are different from v to some set of predicates. Let $Q_1, Q_2 \subseteq Q(v)$. Then, v is a positive unknown if

$$\forall \sigma_v, Q_1, Q_2 : (Q_1 \Rightarrow Q_2) \quad \Rightarrow \quad (\tau\sigma_v[v \mapsto Q_1] \Rightarrow \tau\sigma_v[v \mapsto Q_2])$$

Similarly, v is a negative unknown if

$$\forall \sigma_v, Q_1, Q_2 : (Q_1 \Rightarrow Q_2) \quad \Rightarrow \quad (\tau\sigma_v[v \mapsto Q_2] \Rightarrow \tau\sigma_v[v \mapsto Q_1])$$

Example 3.1 *Consider the template $\tau \doteq v_1 \Rightarrow v_2$. Let us see how v_1 is a negative unknown while v_2 is a positive unknown in τ. Let σ_v be some arbitrary map, e.g., $\sigma_v = \{v_1 \mapsto x > 0\}$. Then $\tau\sigma_v$ evaluates to $x > 0 \Rightarrow v_2$. For v_2 to be a positive variable in τ, then it must satisfy*

$$\forall Q_1, Q_2 : (Q_1 \Rightarrow Q_2) \quad \Rightarrow \quad ((x > 0 \Rightarrow v_2)[v_2 \mapsto Q_1] \Rightarrow (x > 0 \Rightarrow v_2)[v_2 \mapsto Q_2])$$

or equivalently,

$$\forall Q_1, Q_2 : (Q_1 \Rightarrow Q_2) \quad \Rightarrow \quad ((x > 0 \Rightarrow Q_1) \Rightarrow (x > 0 \Rightarrow Q_2))$$

The consequent simplifies to $(x > 0 \wedge \neg Q_1) \vee (\neg(x > 0) \vee Q_2)$. By distributing the disjunction over the conjunction in the first term and simplifying, this reduces to $\neg(x > 0) \vee \neg Q_1 \vee Q_2$. This is the same as $x > 0 \Rightarrow (Q_1 \Rightarrow Q_2)$, which trivially holds under the antecedent $Q_1 \Rightarrow Q_2$. An analogous argument shows that v_1 is a negative unknown.

If each unknown variable in a template/formula occurs only once, then it is easy to see each unknown is either positive or negative. We use the notation $\texttt{Unk}^+(\tau)$ and $\texttt{Unk}^-(\tau)$ to denote the set of all positive unknowns and negative unknowns respectively in τ. The sets $\texttt{Unk}^+(\tau)$ and $\texttt{Unk}^-(\tau)$ can be computed using structural decomposition of τ as shown in Figure 3.5.

$$\begin{array}{rcl|rcl}
\mathtt{Unk}^+(v) & = & \{v\} & \mathtt{Unk}^-(v) & = & \emptyset \\
\mathtt{Unk}^+(\neg\tau) & = & \mathtt{Unk}^-(\tau) & \mathtt{Unk}^-(\neg\tau) & = & \mathtt{Unk}^+(\tau) \\
\mathtt{Unk}^+(\tau_1 \wedge \tau_2) & = & \mathtt{Unk}^+(\tau_1) \cup \mathtt{Unk}^+(\tau_2) & \mathtt{Unk}^-(\tau_1 \wedge \tau_2) & = & \mathtt{Unk}^-(\tau_1) \cup \mathtt{Unk}^-(\tau_2) \\
\mathtt{Unk}^+(\tau_1 \vee \tau_2) & = & \mathtt{Unk}^+(\tau_1) \cup \mathtt{Unk}^+(\tau_2) & \mathtt{Unk}^-(\tau_1 \vee \tau_2) & = & \mathtt{Unk}^-(\tau_1) \cup \mathtt{Unk}^-(\tau_2) \\
\mathtt{Unk}^+(\forall X : \tau) & = & \mathtt{Unk}^+(\tau) & \mathtt{Unk}^-(\forall X : \tau) & = & \mathtt{Unk}^-(\tau) \\
\mathtt{Unk}^+(\exists X : \tau) & = & \mathtt{Unk}^+(\tau) & \mathtt{Unk}^-(\exists X : \tau) & = & \mathtt{Unk}^-(\tau)
\end{array}$$

Figure 3.5: Structural decomposition of a formula τ to compute the set of positive ($\mathtt{Unk}^+(\tau)$) and negative ($\mathtt{Unk}^-(\tau)$) unknowns.

Example 3.2 *Consider the following template τ with unknown variables $v_1, \ldots, v_5$.*

$$(v_1 \;\wedge\; (\forall j : v_2 \Rightarrow \mathtt{sel}(A,j) \leq \mathtt{sel}(B,j)) \;\wedge$$

$$(\forall j : v_3 \Rightarrow \mathtt{sel}(B,j) \leq \mathtt{sel}(C,j))) \;\;\Rightarrow$$

$$(v_4 \;\wedge\; (\forall j : v_5 \Rightarrow \mathtt{sel}(A,j) \leq \mathtt{sel}(C,j)))$$

Then, $\mathtt{Unk}^+(\tau) = \{v_2, v_3, v_4\}$ *and* $\mathtt{Unk}^-(\tau) = \{v_1, v_5\}$. *Note our modeling of arrays using select (*`sel`*) predicates as described in the next section.*

3.3.2 Program Model

We assume that a program `Prog` consists of the following kind of statements s (besides the control-flow).

$$s \;::=\; x := e \;\mid\; \mathtt{assert}(\phi) \;\mid\; \mathtt{assume}(\phi)$$

In the above, x denotes a variable and e denotes some expression. Memory reads and writes can be modeled using memory variables, e.g., variables denoting arrays, and using McCarthy's select (`sel`) and update (`upd`) predicates [198]. Since we allow for `assume` statements, without loss of generality we can treat all conditionals in the program as non-deterministic.

We now give a formalism in which different templates can be associated with different program points, and different unknowns in templates can take values from different sets of predicates. Recall from Chapter 2 that a cut-set C of a program `Prog` is a set of program points, called cut-points, such that any cyclic path in `Prog` passes through some cut-point. Every cut-point in C is labeled with an invariant template. For simplicity, we assume that C also consists of program entry and exit locations, which are labeled with an invariant template that is simply *true*. Let `Paths`(`Prog`) denote the set of all tuples $(\delta, \tau_1, \tau_2, \sigma_t)$, where δ is some straight-line path between two cut-points from C that are labeled with invariant templates τ_1 and τ_2 respectively. Without loss of any generality, we assume that each program path δ is in static single assignment (SSA) form. The variables that are live at start of path δ are the original program variables, and the SSA versions of the variables that are live at the end of δ are given by a map $\sigma_t \doteq \{v_i \mapsto v'_i\}_i$, while $\sigma_t^{-1} \doteq \{v'_i \mapsto v_i\}_i$ denotes the reverse map, where v_i and v'_i are the corresponding variables live at the beginning and end, respectively.

Notice that in the previous chapter we did not make this assumption about the program being in SSA form. We will see later that SSA form allows us to treat predicates opaquely, as is required here, while in the previous chapter we could inspect, and substitute into, the linear relations.

We use the notation `Unk`(`Prog`) to denote the set of unknown variables in the invariant templates at all cut-points of `Prog`.

Example 3.3 *Consider as a running example the program* `ArrayInit` *below, which* *initializes all array elements to* 0*. Consider for this program, a cut-set* C *that consists of only the program location* 2*, besides the entry location and the exit location. Let the program location* 2 *be labeled with the invariant template* $\forall j : v \Rightarrow \texttt{sel}(A, j) = 0$*, which has one negative unknown* v*. Then,* $\texttt{Paths}(\texttt{ArrayInit})$ *consists of the following tuples.*

```
  ArrayInit(int* $A$, int $n$)
1   $i := 0$;
2   while $(i < n)$
3         $A[i] := 0$;
4         $i := i + 1$;
5   Assert($\forall j : 0 \leq j < n \Rightarrow \texttt{sel}(A, j) = 0$);
```

Entry Case $(i := 0, \mathit{true}, \forall j : v \Rightarrow \texttt{sel}(A, j) = 0, \sigma_t)$*, where* σ_t *is the identity map.*

Exit Case $(\mathit{assume}(i \geq n), \forall j : v \Rightarrow \texttt{sel}(A, j) = 0, \forall j : 0 \leq j < n \Rightarrow \texttt{sel}(A, j) = 0, \sigma_t)$*, where* σ_t *is the identity map.*

Inductive Case $(\mathit{assume}(i < n); A' := \texttt{upd}(A, i, 0); i' := i + 1, \forall j : v \Rightarrow \texttt{sel}(A, j) = 0, \forall j : v \Rightarrow \texttt{sel}(A', j) = 0, \sigma_t)$*, where* $\sigma_t(i) = i', \sigma_t(A) = A'$*.*

3.3.3 Invariant Solution

In Section 2.2.1, we reviewed verification conditions. We will use the same framework in this chapter, but it is important to revisit the definition as we will use a slightly different mechanism for reasoning about assignments (as hinted earlier).

We will now define a verification condition as parameterized by the straight-line path δ (a sequence of statements s) in SSA form between two program points and by the invariant templates τ_1 and τ_2 at those points, as follows:

$$\texttt{VC}(\langle \tau_1, \delta, \tau_2 \rangle) \;\; = \;\; \tau_1 \Rightarrow \texttt{WP}(\delta, \tau_2)$$

The weakest liberal precondition $\texttt{WP}(\delta, \phi)$ of formula ϕ with respect to path δ is almost as before, restated in Table 3.1, *except* for the difference in the handling of assignment. An assignment is now translated to an equality predicate. Observe that the correctness of the assignment rule in Table 3.1 relies on the fact that the statements on path δ are in SSA form. This is important since otherwise we will have to address the issue of substitution in templates, as the only choice for $\texttt{WP}(x := e, \phi)$ when the path δ is in non-SSA form would be $\phi[e/x]$. In this chapter, our algorithms treat predicates opaquely (as long as the SMT solver understands their interpretation), and consequently substitution is not a viable option.

$$\begin{array}{rcl}
\texttt{WP}(\texttt{skip}, \phi) & = & \phi \\
\texttt{WP}(s_1; s_2, \phi) & = & \texttt{WP}(s_1, \texttt{WP}(s_2, \phi)) \\
\texttt{WP}(\texttt{assert}(\phi'), \phi) & = & \phi' \wedge \phi \\
\texttt{WP}(\texttt{assume}(\phi'), \phi) & = & \phi' \Rightarrow \phi \\
\texttt{WP}(x := e, \phi) & = & (x = e) \Rightarrow \phi
\end{array}$$

Table 3.1: Weakest precondition transformer.

Definition 3.1 (Invariant Solution) *Let Q be a* predicate-map *that maps each unknown v in any template invariant in program* `Prog` *to some set of predicates $Q(v)$. Let σ map each unknown v in any template invariant in program* `Prog` *to some subset of $Q(v)$. We say that σ is an* invariant solution *for* `Prog` *over Q if the following formula* $\texttt{VC}(\texttt{Prog}, \sigma)$, *which denotes the verification condition of the* *program* `Prog` *w.r.t.* σ*, is valid.*

$$\mathtt{VC}(\mathtt{Prog},\sigma) \overset{def}{=} \bigwedge_{(\delta,\tau_1,\tau_2,\sigma_t)\in\mathtt{Paths}(\mathtt{Prog})} \mathtt{VC}(\langle\tau_1\sigma,\delta,\tau_2\sigma\sigma_t\rangle)$$

Example 3.4 *Consider the program* `ArrayInit` *described in Example 3.3. Let* Q *map unknown* v *in the invariant template at cut-point location* 2 *to* $Q_{j,\{0,i,j\}}$*. Let* σ *map* v *to* $Q_0 = \{0 \le j, j < i\}$*. Then,* σ *is an invariant solution for* `ArrayInit` *over* Q *since the verification condition* $\mathtt{VC}(\mathtt{ArrayInit},\sigma)$ *of the program* `ArrayInit`*, which is given by the conjunction of the following formulas, is valid.*

- $i = 0 \;\;\Rightarrow\;\; (\forall j : Q_0 \Rightarrow \mathtt{sel}(A,j) = 0)$
- $(i \ge n \;\;\wedge\;\; (\forall j : Q_0 \Rightarrow \mathtt{sel}(A,j) = 0)) \;\;\Rightarrow (\forall j : 0 \le j \le n \Rightarrow \mathtt{sel}(A,j) = 0)$
- $(i < n \;\wedge\; A' = \mathtt{upd}(A,i,0) \;\wedge\; i' = i+1 \;\wedge$

  $$(\forall j : Q_0 \Rightarrow \mathtt{sel}(A,j) = 0)) \;\;\Rightarrow (\forall j : Q_0\sigma_t \Rightarrow \mathtt{sel}(A',j) = 0)$$

  where $\sigma_t(i) = i'$ *and* $\sigma_t(A) = A'$*.*

Sections 3.5 and 3.6 describe algorithms for generating an invariant solution given program `Prog` and an appropriate predicate-map Q.

3.4 Optimal Solutions

In this section, we present the core operation of generating an *optimal solution* that is used by our algorithm to perform local reasoning about program paths, which are encoded as formulae. Separating local reasoning from fixed-point computation is essential because the semantics of a program with loops cannot be exactly encoded as an SMT constraint.

```
OptimalSolutions($\phi, Q$)
Let Unk$^+(\phi)$ be $\{\rho_1, .., \rho_a\}$.
Let Unk$^-(\phi)$ be $\{\eta_1, .., \eta_b\}$.
$S := \emptyset$;
foreach $\langle q_1, .., q_a \rangle \in Q(\rho_1) \times .. \times Q(\rho_a)$:
    $\phi' := \phi[\rho_i \mapsto \{q_i\}]_i$;
    $T :=$ OptimalNegativeSolutions($\phi', Q$);
    $S := S \cup \{\sigma \mid \sigma(\rho_i) = \{q_i\}, \sigma(\eta_i) = t(\eta_i), t \in T\}$;
$R := \{$MakeOptimal($\sigma, S$) $\mid \sigma \in S\}$;
$R :=$ Saturate($R, S$);
return $R$;
```

```
Saturate($R, S$)
while any change in $R$:
    foreach $\sigma_1, \sigma_2 \in R$
        $\sigma :=$ Merge($\sigma_1, \sigma_2, S$); if $(\sigma = \bot)$ continue;
        if $\nexists \sigma' \in R : \bigwedge_{i=1}^{a} \sigma'(\rho_i) \Rightarrow \sigma(\rho_i) \wedge \bigwedge_{i=1}^{b} \sigma(\eta_i) \Rightarrow \sigma'(\eta_i)$
            $R := R \cup \{$MakeOptimal($\sigma, S$)$\}$;
return $R$;
```

```
MakeOptimal($\sigma, S$)
$T := \{\sigma' \mid \sigma' \in S \wedge \bigwedge_{i=1}^{b} \sigma(\eta_i) \Rightarrow \sigma'(\eta_i)\}$
foreach $\sigma' \in T$:
    $\sigma'' :=$ Merge($\sigma, \sigma', S$)
    if $(\sigma'' \neq \bot)$ $\sigma := \sigma''$;
return $\sigma$
```

```
Merge($\sigma_1, \sigma_2, S$)
Let $\sigma$ be s.t. $\sigma(\rho_i) = \sigma_1(\rho_i) \cup \sigma_2(\rho_i)$ for $i$ = 1 to $a$
            and $\sigma(\eta_i) = \sigma_1(\eta_i) \cup \sigma_2(\eta_i)$ for $i$ = 1 to $b$
$T := \{\sigma' \mid \sigma' \in S \wedge \bigwedge_{i=1}^{b} \sigma(\eta_i) \Rightarrow \sigma'(\eta_i)\}$
if $\bigwedge_{q_1 \in \sigma(\rho_1), .., q_a \in \sigma(\rho_a)} \exists \sigma' \in T$ s.t. $\bigwedge_{i=1}^{a} \sigma'(\rho_i) = \{q_i\}$ return $\sigma$
else return $\bot$
```

Figure 3.6: Procedure for generating optimal solutions given a template formula ϕ and a predicate-map Q.

Semantics of loopy programs as opposed to SMT

Encoding the semantics of programs with loops would mean being able to solve for the invariant solution from Definition 3.1; which is the implicitly quantified formula $\exists \sigma \forall X : \texttt{VC}(\texttt{Prog}, \sigma)$, where X is the set of program variables that appear in the verification condition. On the other hand an SMT formula ϕ that we have solvers for are implicitly quantified as $\exists X' : \phi$, where X' is the set of variables that appear in ϕ. Notice, that because of the quantifier alternation in the first formula, it cannot be manipulated such that it is directly an SMT query, which has no quantifier alternation. However, the results in this chapter, demonstrate that SMT queries can be used to gather enough information such that we can infer the required σ using an efficient algorithm.

We will discuss fixed-point computation using the information derived from the local reasoning technique developed here in Sections 3.5 and 3.6.

Definition 3.2 (Optimal Solution) *Let ϕ be a formula with unknowns $\{v_i\}_i$ where each v_i is either positive or negative. Let Q map each unknown v_i to some set of predicates $Q(v_i)$. A map $\{v_i \mapsto Q_i\}_i$ is a* solution *(for ϕ over domain Q) if the formula ϕ is valid after each v_i is replaced by Q_i, and $Q_i \subseteq Q(v_i)$. A solution $\{v_i \mapsto Q_i\}_i$ is* optimal *if replacing Q_i by a strictly weaker or stronger subset of predicates from $Q(v_i)$, for the case where v_i is negative or positive, respectively, results in a map that is no longer a solution.*

Example 3.5 *Consider the following formula ϕ with one negative unknown η.*

$$i = 0 \;\Rightarrow\; (\forall j : \eta \Rightarrow \texttt{sel}(A, j) = 0)$$

Let $Q(\eta)$ be $Q_{j,\{0,i,n\}}$. There are four optimal solutions for ϕ over Q. These map the negative unknown variable η to $\{0 < j \leq i\}$, $\{0 \leq j < i\}$, $\{i < j \leq 0\}$, and $\{i \leq j < 0\}$ respectively.

Since the naive exponential search for optimal solutions to a formula would be too expensive, we next present a systematic search that we found to be efficient in practice.

The procedure described in Figure 3.6 returns the set of all optimal solutions for an input formula ϕ over domain Q. The procedure `OptimalSolutions` uses an operation `OptimalNegativeSolutions`(ϕ, Q) (discussed later), which returns the set of all optimal solutions for the special case when ϕ consists of only negative unknowns. To understand how the procedure `OptimalSolutions` operates, it is illustrative to think of the simple case when there is only one positive variable ρ. In this case, the algorithm simply returns the conjunction of all those predicates $q \in Q(\rho)$ such that $\phi[\rho \mapsto \{q\}]$ is valid. Observe that such a solution is an optimal solution, and this procedure is much more efficient than naively trying out all possible subsets and picking the maximal ones.

Example 3.6 *Consider the following formula ϕ with one positive unknown ρ.*

$$(i \geq n) \wedge (\forall j : \rho \Rightarrow \texttt{sel}(A, j) = 0)) \;\Rightarrow$$
$$(\forall j : 0 \leq j < n \Rightarrow \texttt{sel}(A, j) = 0)$$

Let $Q(\rho)$ be $Q_{j,\{0,i,n\}}$. There is one optimal solution for ϕ over Q, namely

$$\rho \mapsto \{0 \leq j, j < n, j < i\}$$

This is computed by the algorithm in Figure 3.6 as follows. At the end of the first loop (Lines 4-7), the set S contains three solutions:

$$\textit{1:}\quad \rho \mapsto \{0 \leq j\}$$

$$\textit{2:}\quad \rho \mapsto \{j < n\}$$

$$\textit{3:}\quad \rho \mapsto \{j < i\}$$

The set R at the end of line 8 contains only one optimal solution:

$$\rho \mapsto \{0 \leq j, j < n, j < i\}$$

The set R is unchanged after the `Saturate` *call, simply because it contains only one optimal solution, while any change to R would require R to contain at least two optimal solutions.*

Now, consider the case of one positive and one negative variable. In this case, the algorithm invokes `OptimalNegativeSolutions` to find an optimal set of negative solutions for the negative variable η, for each choice of predicate $q \in Q(\rho)$ for the positive variable ρ, and stores these solutions in set S (Lines 4-7). After this, it groups together all those solutions in S that match on the negative variable to generate a set R of optimal solutions (Line 8). (Recall, from Definition 3.2, that in an optimal solution a positive variable is mapped to a maximal set of predicates, while a negative variable is mapped to a minimal set.) It then attempts to generate more optimal solutions by merging the solutions for *both* the positive and negative variables of the optimal solutions in R through the call to `Saturate` (Line 9).

Example 3.7 *Consider the following formula ϕ with one positive unknown ρ and one negative unknown η.*

$$(\eta \wedge (i \geq n) \wedge (\forall j : \rho \Rightarrow \texttt{sel}(A, j) = 0)) \;\Rightarrow$$

$$(\forall j : j \leq m \Rightarrow \texttt{sel}(A, j) = 0)$$

Let $Q(\eta)$ and $Q(\rho)$ both be $Q_{\{i,j,n,m\}}$. There are three optimal solutions for ϕ over Q, namely

$$\begin{array}{lll} \textit{1:} & \rho \mapsto \{j \leq m\} & , \quad \eta \mapsto \emptyset \\ \textit{2:} & \rho \mapsto \{j \leq n, j \leq m, j \leq i\}, & \eta \mapsto \{m \leq n\} \\ \textit{3:} & \rho \mapsto \{j \leq i, j \leq m\} & , \quad \eta \mapsto \{m \leq i\} \end{array}$$

These are computed by the algorithm in Figure 3.6 as follows. At the end of the first loop (Lines 4-7), the set S contains the following four solutions:

$$\begin{array}{lll} \textit{1:} & \rho \mapsto \{j \leq m\}, & \eta \mapsto \emptyset \\ \textit{2:} & \rho \mapsto \{j \leq n\}\ , & \eta \mapsto \{m \leq n\} \\ \textit{3:} & \rho \mapsto \{j \leq i\}\ \ , & \eta \mapsto \{m \leq i\} \\ \textit{4:} & \rho \mapsto \{j \leq i\}\ \ , & \eta \mapsto \{m \leq n\} \end{array}$$

The set R at the end of line 8 contains the following three optimal solutions:

$$\begin{array}{lll} \textit{1:} & \rho \mapsto \{j \leq m\} & , \quad \eta \mapsto \emptyset \\ \textit{2:} & \rho \mapsto \{j \leq n, j \leq m, j \leq i\}, & \eta \mapsto \{m \leq n\} \\ \textit{3:} & \rho \mapsto \{j \leq i, j \leq m\} & , \quad \eta \mapsto \{m \leq i\} \end{array}$$

The set R is unchanged by the call to `Saturate` *(Line 9).*

The extension to multiple positive variables involves considering a choice of all tuples of predicates of appropriate size (Line 4), while the extension to multiple negative variables is not very different.

The proof of correctness of the `OptimalSolutions` procedure described here is given in Appendix A.3, and we encourage the reader to go through it to get a better understanding of the working of the procedure.

The `OptimalNegativeSolutions` *operation* This operation requires reasoning over the theories that are used in the predicates, e.g., the theory of arrays, the bit vector theory, or linear arithmetic. We use an SMT solver as a black box for such theory reasoning. Of several ways to implement `OptimalNegativeSolutions`, we found it effective to implement `OptimalNegativeSolutions`(ϕ, Q) as a breadth-first search on the lattice of subsets ordered by implication, with $\top$ and $\bot$ being $\emptyset$ and the set of all predicates, respectively. We start at $\top$ and keep deleting the subtree of every solution discovered until no more elements remain to be searched. Furthermore, to achieve efficiency, one can truncate the search at a certain depth. (We observed that the number of predicates mapped to a negative variable in any optimal solution in our experiments was never greater than 4.) To achieve completeness, the bounding depth can be increased iteratively after a failed attempt.

`OptimalNegativeSolutions` *and predicate cover* The operation `OptimalNegativeSolutions` as we define above is a generalization of the predicate cover operation from standard predicate abstraction literature [128, 177]. Given a set of predicates Q_0 and a formula ϕ, the predicate cover operation finds the weakest conjunction of predicates from Q_0 that implies it. This is illustrated pictorially in Figure 3.7. Predicate cover is a fundamental operation used in the abstract transformers while

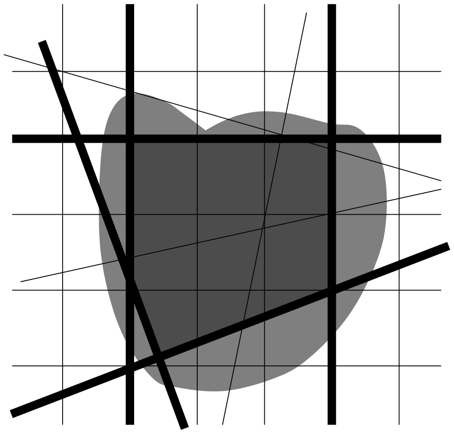


Figure 3.7: The predicate cover operation. The lines indicate predicates and their negations. Pictorially, a predicate specifies one half-space and its negation the other half-space. For a given formula—the light gray area—the predicate cover computed is the set of predicates corresponding to the bold lines. The enclosed space by the predicate cover—the dark gray area—lies completely within area for the formula, indicating that the predicate cover implies the formula. Notice that the computed predicate cover is the maximally weak possible over these predicates: leaving out any predicate/line from the cover will merge areas outside of the formula. Notice that in general there may be multiple maximally weak formulas and it is expected that the predicate cover/`OptimalNegativeSolutions` procedure will output all incomparable ones.

performing abstract interpretation over predicate abstraction [128]. The weakest conjunction corresponds to the least number of predicates.

Note that this is exactly the output of `OptimalNegativeSolutions`$((\eta \Rightarrow \phi), \{Q_0\})$. Since we deal with more general templates, i.e., with arbitrary boolean structure as opposed to just conjunctive facts as in previous predicate abstraction literature, we need to generalize through `OptimalNegativeSolutions` the notion of predicate cover to handle multiple negative unknowns. Additionally, we also need to build another operation `OptimalSolutions` to handle positive unknowns as well.

The proof of correctness of the `OptimalNegativeSolutions` procedure described here is again given in Appendix A.3, and we encourage the reader to go through it to get a better understanding of the design here.

In the following sections we use this `OptimalSolutions` interface to the SMT solver to build fixed-point computation algorithms, two that iteratively approximate the solution (Section 3.5) similar to traditional dataflow approaches and one that uses an encoding of the fixed-point as a SAT formula (Section 3.6) similar to the approach in the previous chapter.

3.5 Iterative Propagation Based Algorithms

In this section, we present two iterative propagation based algorithms for discovering an inductive invariant that establishes the validity of assertions in a given program.

The key insight behind these algorithms is as follows. Observe that the set of elements that are instantiations of a given template with respect to a given set of predicates, ordered by implication, forms a pre-order, but not a lattice. Our algorithms perform a standard data-flow analysis over the powerset extension of this abstract domain (which forms a lattice) to ensure that it does not miss any solution. Experimental evidence shows that the number of elements in this powerset extension never gets beyond 6. Each step in the algorithm involves updating a fact at a cut-point by using the facts at the neighboring cut-points (preceding or succeeding cut-points in case of forward or backward data-flow, respectively). The update is done by generating the verification condition that relates the facts at the neighboring cut-points with the template at the current cut-point, and updating

```
LeastFixedPoint(Prog, $Q$)
  Let $\sigma_0$ be s.t. $\sigma_0(v) \mapsto \emptyset$, if $v$ is negative
                         $\sigma_0(v) \mapsto Q(v)$, if $v$ is positive
  $S := \{\sigma_0\}$;
  while $S \neq \emptyset \wedge \forall \sigma \in S : \neg\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$
      Choose $\sigma \in S, (\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$ s.t.
                $\neg\texttt{Valid}(\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t \rangle))$
      $S := S - \{\sigma\}$;
      Let $\sigma_p = \sigma \mid_{\texttt{Unk(Prog)}-\texttt{Unk}(\tau_2)}$ and $\theta := \tau_2\sigma \Rightarrow \tau_2$.
      $S := S \cup \{\sigma'\sigma_t^{-1} \cup \sigma_p \mid \bigwedge_{\sigma'' \in S} \tau_2\sigma'' \not\Rightarrow \tau_2\sigma'\sigma_t^{-1} \ \wedge$
           $\sigma' \in \texttt{OptimalSolutions}(\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2 \rangle) \wedge \theta, Q\sigma_t)\}$
  if $S = \emptyset$ return ``No solution''
  else return $\sigma \in S$ s.t. $\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$
```

(a) Least Fixed-Point Computation

```
GreatestFixedPoint(Prog)
  Let $\sigma_0$ be s.t. $\sigma_0(v) \mapsto Q(v)$, if $v$ is negative
                         $\sigma_0(v) \mapsto \emptyset$, if $v$ is positive
  $S := \{\sigma_0\}$;
  while $S \neq \emptyset \wedge \forall \sigma \in S : \neg\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$
      Choose $\sigma \in S, (\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$ s.t.
                $\neg\texttt{Valid}(\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t \rangle))$
      $S := S - \{\sigma\}$;
      Let $\sigma_p = \sigma \mid_{\texttt{Unk(Prog)}-\texttt{Unk}(\tau_1)}$ and $\theta := \tau_1 \Rightarrow \tau_1\sigma$.
      $S := S \cup \{\sigma' \cup \sigma_p \mid \bigwedge_{\sigma'' \in S} \tau_1\sigma' \not\Rightarrow \tau_1\sigma'' \ \wedge$
           $\sigma' \in \texttt{OptimalSolutions}(\texttt{VC}(\langle \tau_1, \delta, \tau_2\sigma\sigma_t \rangle) \wedge \theta, Q)\}$
  if $S = \emptyset$ return ``No solution''
  else return $\sigma \in S$ s.t. $\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$
```

(b) Greatest Fixed-Point Computation

Figure 3.8: Iterative algorithms for generating an invariant solution given program `Prog` and predicate-map Q.

using the solutions obtained from a call to `OptimalSolutions`.

The two algorithms differ in whether they perform a forward or backward dataflow and accordingly end up computing a least or greatest fixed point, respectively, but they both have the following property.

Theorem 3.1 (Correctness of Iterative Fixed-point Computation) *Given a program* `Prog` *and a predicate map* Q*, the algorithms in Figure 3.8 output an invariant solution, if there exists one.*

For notational convenience, we present the algorithms slightly differently. Each of these algorithms (described in Figure 3.8) involve maintaining a set of candidate solutions at each step. A *candidate solution* σ is a map of the unknowns v in all templates to some subset of $Q(v)$, where Q is the given predicate-map. The algorithms make progress by choosing a candidate solution and replacing it by a set of weaker or stronger candidate solutions (depending on whether a forward/least fixed-point or backward/greatest fixed-point technique is used) using the operation `OptimalSolutions` defined in Section 3.4. The algorithms return an invariant solution whenever any candidate solution σ satisfies the verification condition, i.e., $\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$, or fail when the set of candidate solutions becomes empty.

The proof of Theorem 3.1 follows directly from the correctness of dataflow analyses [164]. The procedure `OptimalSolutions` serves as both the forward and backwards transfer function by computing the optimal change that is required to the invariant at the endpoint of a path (Theorem A.3). The fixed-point algorithms (Figure 3.8) implement a iterative work-list dataflow computation. The lattice is the finite height lattice of maps ordered by the partial order $\sqsubseteq$ as defined below. Line 7 in Figure 3.8(a) and Line 7 in Figure 3.8(b) implement the join operation.

Definition 3.3 (Ordering $\sqsubseteq$ of solutions) *Given a template τ, two solutions σ_1 and σ_2 are ordered as $\sigma_1 \sqsubseteq \sigma_2$ iff $\forall \rho \in \texttt{Unk}^+(\tau) : \sigma_1[\rho] \Rightarrow \sigma_2[\rho]$ and $\forall \eta \in \texttt{Unk}^-(\tau) : \sigma_2[\eta] \Rightarrow \sigma_1[\eta]$.*

We next discuss the two variants for computing least and greatest fixed-points, along with an example.

3.5.1 Least Fixed-point

We now describe a least fixed-point approach that starts at the bottom of the lattice, and refines the invariants to a weaker one in each iteration. It iterates until the candidate solution is weak enough to be valid for given the precondition.

This algorithm (Figure 3.8(a)) starts with the singleton set containing the candidate solution that maps each negative unknown to the empty set (i.e., *true*) and each positive unknown to the set of all predicates. In each step, the algorithm chooses a σ that is not an invariant solution. Since it is not an invariant solution, it must be the case that it does not satisfy at least one verification condition. There must exist a $(\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$ such that $\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t \rangle)$ is not valid, because $\tau_2\sigma$ is a too strong an instantiation for τ_2. (This is because the loop on Line 3 in the algorithm maintains the invariant that any assignment to τ_2 at the end of a verification condition is at least as strong as it can be given the verification condition and the assignment to τ_1 at its beginning.) The algorithm replaces the candidate solution σ by the solutions $\{\sigma'\sigma_t^{-1} \cup \sigma_p \mid \sigma' \in \texttt{OptimalSolutions}(\texttt{VC}(\langle\tau_1\sigma, \delta, \tau_2\rangle) \wedge \theta, Q\sigma_t)\}$, where σ_p is the projection of the map σ onto the unknowns in the set $\texttt{Unk}(\texttt{Prog}) - \texttt{Unk}(\tau_2)$ and θ (defined as $\tau_2\sigma \Rightarrow \tau_2$) ensures that only stronger solutions are considered.

Example 3.8 *Consider the* `ArrayInit` *program from Example 3.3. Let* $Q(v) = Q_{j,\{0,i,n\}}$. *In the first iteration of the while loop, S is initialized to σ_0, and in Line 4 there is only one triple in* `Paths(ArrayInit)` *whose corresponding verification condition is inconsistent, namely* $(i := 0, true, \forall j : v \Rightarrow \texttt{sel}(A, j) = 0, \sigma_t)$, *where σ_t is the identity map. Line 7 results in a call to* `OptimalSolutions` *on the formula* $\phi = (i = 0) \Rightarrow (\forall j : v \Rightarrow \texttt{sel}(A, j) = 0)$, *the result of which has already been shown in Example 3.5. The set S now contains the following candidate solutions after the first iteration of the while loop.*

1: $v \mapsto \{0 < j \leq i\}$

2: $v \mapsto \{0 \leq j < i\}$

3: $v \mapsto \{i < j \leq 0\}$

4: $v \mapsto \{i \leq j < 0\}$

Of these, the candidate solution $v \mapsto \{0 \leq j < i\}$ *is a valid solution, and hence the while loop terminates after one iteration.*

3.5.2 Greatest Fixed-point

Similar to the least fixed-point computation in the previous section, we now present a greatest fixed-point approach. The key difference is that instead of starting the iteration from the bottom of the lattice, we instead start at the top and refine the invariants to a stronger one in each iteration. It iterates until the candidate solution is strong enough to imply the postconditions. We detail the approach here for completeness.

This algorithm (Figure 3.8(b)) starts with the singleton set containing the candidate solution that maps each positive unknown to the empty set (i.e., *true*) and each negative unknown to the set of all predicates. As above, in each step the algorithm chooses a σ that is not an invariant solution. Since it is not an invariant solution, it must be the case that it does not satisfy at least one verification condition. There must exist a $(\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$ such that $\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t \rangle)$ is not valid, because $\tau_1\sigma$ is a too weak an instantiation for τ_1. (This is because the loop on Line 3 in the algorithm maintains the invariant that any assignment to τ_1 at the beginning of a verification condition is at least as weak as it can be given the verification condition and the assignment to τ_2 at its end.) The algorithm replaces the candidate solution σ by the solutions $\{\sigma' \cup \sigma_p \mid \sigma' \in \texttt{OptimalSolutions}(\texttt{VC}(\langle \tau_1, \delta, \tau_2\sigma\sigma_t \rangle) \wedge \theta, Q)\}$, where σ_p is the projection of the map σ onto the unknowns in the set $\texttt{Unk}(\texttt{Prog}) - \texttt{Unk}(\tau_1)$ and θ (defined as $\tau_1 \Rightarrow \tau_1\sigma$) ensures that only weaker solutions are considered.

Example 3.9 *Consider the* `ArrayInit` *program from Example 3.3. Let* $Q(v) = Q_{j,\{0,i,n\}}$*. In the first iteration of the while loop, S is initialized to* σ_0*, and in Line 4 there is only one triple in* `Paths(ArrayInit)` *whose corresponding verification condition is inconsistent, namely* $(\texttt{assume}(i \geq n), \forall j : v \Rightarrow \texttt{sel}(A, j) = 0, \forall j : 0 \leq j <$

$n \Rightarrow$ `sel`$(A, j) = 0, \sigma_t)$, where σ_t is the identity map. Line 7 results in a call to `OptimalSolutions` *on the formula $\phi = (i \geq n) \wedge (\forall j : v \Rightarrow$ `sel`$(A, j) = 0) \Rightarrow (\forall j : 0 \leq j < n \Rightarrow$ `sel`$(A, j) = 0)$, whose output is shown in Example 3.6. This results in S containing only the following candidate solution after the first iteration of the while loop:*

$$v \mapsto \{0 \leq j, j < n, j < i\}$$

The candidate solution $v \mapsto \{0 \leq j, j < n, j < i\}$ is a valid solution, and hence the while loop terminates after one iteration.

3.6 Satisfiability-based Algorithm

In this section, we show how to encode the verification condition of the program as a boolean formula such that a satisfying assignment to the boolean formula corresponds to an inductive invariant that establishes the validity of assertions in a given program. We describe how verification conditions can be reduced to propositional constraints in two steps. We first describe the simpler case of just conjunctive invariants (or k disjuncts each being conjunctive) in Section 3.6.1 and then step up to an efficient reduction for arbitrary templates using `OptimalNegativeSolutions` in Section 3.6.2.

3.6.1 SAT Encoding for Simple Templates

We first illustrate our approach by means of a simple example that discovers a single conjunctive fact I and later extend that to boolean constraint generation for DNF formulae with k disjuncts each, i.e., k-DNF.

Example Consider the program in Figure 3.9(a). The program loop iterates using the loop counter x and increments an auxiliary variable y as well. Its control flow graph (CFG) is shown in Figure 3.9(b), and its equivalent using only non-deterministic branches, assumes, asserts, and assignments is shown in Figure 3.9(c). There are three simple paths going from program entry to loop header ($\boxed{1} \rightarrow \boxed{2}$), around the loop ($\boxed{2} \rightarrow \boxed{2}$), and loop header to program exit ($\boxed{2} \rightarrow \boxed{3}$), and the verification conditions they generate are shown in Figure 3.9(d). The set of predicates $Q(I)$ over which we seek to discover our inductive invariant is shown in Figure 3.9(e).

The first step is to associate with each predicate $p \in Q(I)$ a boolean indicator variable b_p indicating p's presence or absence in I. Then we consider each verification condition for each path in turn and generate constraints on the indicator variables:

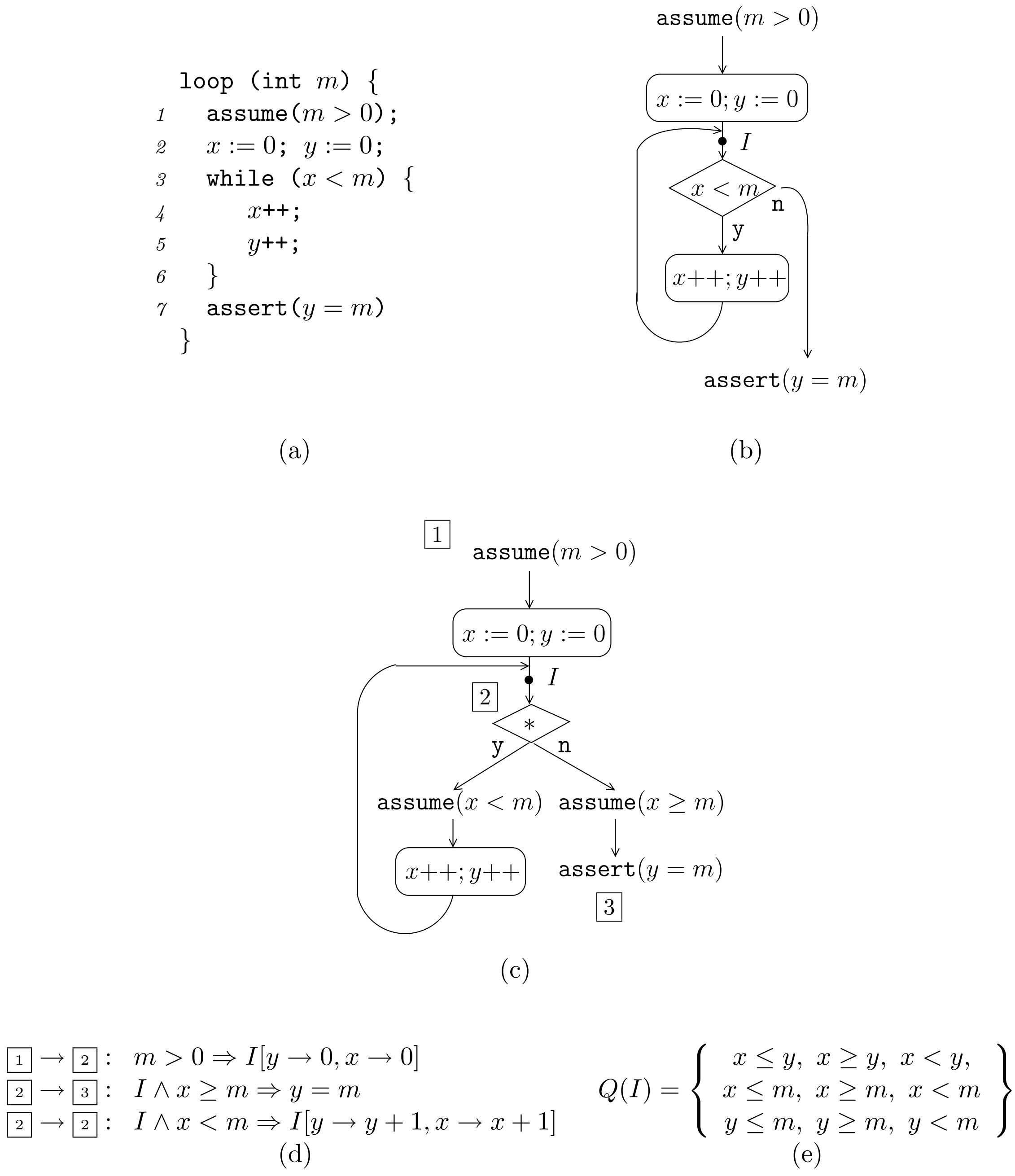
- *Loop entry* ($\boxed{1} \rightarrow \boxed{2}$): The verification condition is $m > 0 \Rightarrow I[y \rightarrow 0, x \rightarrow 0]$, for which we generate the constraint

  $$\neg b_{x<y} \wedge \neg b_{x \geq m} \wedge \neg b_{y \geq m} \tag{Ex-1}$$

  denoting that the predicates $x < y$ and $x \geq m$ and $y \geq m$ cannot be in I since they are not implied by the verification condition for loop entry.

- *Loop exit* ($\boxed{2} \rightarrow \boxed{3}$): The verification condition is $I \wedge x \geq m \Rightarrow y = m$, for which we generate the constraint

  $$(b_{y \geq m} \wedge b_{y \leq m}) \vee b_{x<m} \vee (b_{x \leq y} \wedge b_{y \leq m}) \tag{Ex-2}$$



```
loop (int m) {
  assume(m > 0);
  x := 0;  y := 0;
  while (x < m) {
      x++;
      y++;
  }
  assert(y = m)
}
```

(a)

(b)

(c)

$$\begin{aligned}
\boxed{1} \rightarrow \boxed{2} &: \quad m > 0 \Rightarrow I[y \rightarrow 0, x \rightarrow 0] \\
\boxed{2} \rightarrow \boxed{3} &: \quad I \wedge x \geq m \Rightarrow y = m \\
\boxed{2} \rightarrow \boxed{2} &: \quad I \wedge x < m \Rightarrow I[y \rightarrow y + 1, x \rightarrow x + 1]
\end{aligned}$$

(d)

$$Q(I) = \left\{ \begin{array}{c} x \leq y,\ x \geq y,\ x < y, \\ x \leq m,\ x \geq m,\ x < m \\ y \leq m,\ y \geq m,\ y < m \end{array} \right\}$$

(e)

Figure 3.9: Illustrative example for satisfiability-based reduction. (a) Iteration over x with an auxiliary variable y (b) The control flow graph (CFG) with the loop invariant marked as I (c) The CFG as modeled in our system. (d) Verification condition corresponding to each simple path. (e) The set of predicates Q.

denoting that either both $y \geq m$ and $y \leq m$ belong to I, or $x < m$ belongs to I, or both $x \leq y$ and $y \leq m$ belong to I. Observe that these are the only three (maximally-weak) ways in which we can prove $y = m$ under the assumption $x \geq m$. Traditionally, these different ways are computed by using the predicate cover operation (which we commented on in Section 3.2).

- *Inductive* ($\boxed{2} \to \boxed{2}$): The verification condition is $I \wedge x < m \Rightarrow I[y \to y+1, x \to x+1]$, for which we generate the constraint

$$(b_{y \leq m} \Rightarrow (b_{y<m} \vee b_{y \leq x})) \wedge \neg b_{x<m} \wedge \neg b_{y<m} \tag{Ex-3}$$

denoting that if $y \leq m$ belongs to I, then either $y < m$ or $x \leq y \wedge y \leq x$ should also belong to I, and that the predicates $x < m$ and $y < m$ cannot be in I. The reader can easily check that this verification condition allows any other predicate p to be in I because $p \wedge x < m \Rightarrow p[y \to y+1, x \to x+1]$.

These constraints are generated by considering each predicate p, finding the *weakest conditions*, as boolean constraints bc^p, over the set of predicates under which $p \wedge x < m \Rightarrow p[y \to y+1, x \to x+1]$ and then generating the constraint that $b_p \Rightarrow bc^p$. For the predicates $x < m$ and $y < m$, the weakest boolean constraint is in fact `false`, and hence we generate the constraints $\neg b_{x<m}$ and $\neg b_{y<m}$. For the predicate $y \leq m$, the weakest boolean constraint is $b_{y<m} \vee b_{y \leq x}$. For all other predicates, it is `true`.

Putting Eq. (Ex-1), (Ex-2), and (Ex-3) together we get a SAT formula over the boolean indicator variables that encodes the verification condition of the program.

The reader can verify that $b_{x\geq y} = b_{x\leq y} = b_{y\leq m} =$ `true` (and all others `false`) is a satisfying solution. This corresponds to I being $(x = y \wedge y \leq m)$.

3.6.1.1 Encoding VCs as SAT for Simple Templates

We now describe a SAT encoding for discovering inductive invariants I^{π} that can be described using a relatively simple k-DNF formula over a given predicate map Q. In the next section, we will describe a reduction for general templates (and we will have to use the more general `OptimalNegativeSolutions` procedure instead of just predicate cover). In the k-DNF case, we can represent an invariant I at program point π by $k \times s$ *boolean indicator variables* $b^{\pi}_{i,p}$ (where $1{\leq}i{\leq}k$, $p \in Q(I)$, $s = |Q(I)|$). The boolean variable $b^{\pi}_{i,p}$ denotes whether predicate p is present in the i^{th} disjunct of the invariant I at program point π, which we indicate as I^{π} here. We show how to encode the verification condition of the program as a boolean formula ψ over the boolean indicator variables $b^{\pi}_{i,p}$. The boolean formula $\psi_{\texttt{Prog}}$ is satisfiable iff there exist inductive invariants (in k-DNF form) strong enough to prove the validity of the assertions.

We first show how to encode the verification condition of any simple path δ as a boolean formula ψ_{δ}. But first, let us observe that the verification condition for any simple path δ between π_1 and π_2 simplifies to the following form:

$$I^{\pi_1} \Rightarrow (G \Rightarrow I^{\pi_2}) \tag{3.1}$$

where and G are known formulas obtained from the predicates that occur on the path δ. For reducing verification condition, the following three cases arise, which we consider in increasing order of difficulty:

Case 1 *(Path between program entry and a cut-point)* The verification condition in Eq. 3.1 simplifies to the following form after substituting $I^{\pi_1} = \texttt{true}$ and expanding I^{π_2} as $\bigvee_{j=1}^{k} I_j^{\pi_2}$, where each $I_j^{\pi_1}$ is conjunction of some predicates from $Q(I^{\pi_1})$.

$$G \Rightarrow \left(\bigvee_{j=1}^{k} I_j^{\pi_2} \right)$$

The above constraint restricts how strong I^{π_2} can be. Essentially, if some selection of predicates $q_1, .., q_k$ are present in each of the disjuncts (i.e., their corresponding indicators $b_{1,q_1}^{\pi_2}, .., b_{1,q_1}^{\pi_2}$ are *true*), then it better be the case that their disjunction is implied by G. Formally, if $q_1 \in I_1^{\pi_2}, \ldots, q_k \in I_k^{\pi_2}$, then it must be the case that $G \Rightarrow \bigvee_{j=1}^{k} q_j$. Hence, we can rewrite the above constraint as:

$$\bigwedge_{p_1,..,p_k \in Q(I^{\pi_2})} \left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow (G \Rightarrow \bigvee_{j=1}^{k} p_j) \right) \qquad (3.2)$$

This can be encoded as the following boolean constraint $\psi(\delta)$ over boolean indicator variables $b_{i,p}^{\pi_2}$.

$$\psi_\delta \quad = \quad \bigwedge_{p_1,..,p_k \in Q} \left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow bval(G, \bigvee_{j=1}^{k} p_j) \right) \qquad (3.3)$$

where $bval(A, B)$ is an indicator function that output the truth value (*true* or *false*) of $A \Rightarrow B$.

Case 2 *(Path between a cut-point and program exit)* The verification condition in Eq. 3.1 simplifies to the following form after substituting $I^{\pi_2} = \texttt{true}$ and expanding I^{π_1} as $\bigvee_{j=1}^{k} I_j^{\pi_1}$, where each $I_j^{\pi_1}$ is conjunction of some predicates from $Q(I^{\pi_1})$.

$$\left(\bigvee_{i=1}^{k} I_i^{\pi_1}\right) \Rightarrow G \quad \text{or, equivalently,} \quad \bigwedge_{i=1}^{k} (I_i^{\pi_1} \Rightarrow G)$$

The above constraint restricts how weak $I_i^{\pi_1}$ can be. We can encode the above constraint as a boolean formula over the variables $b_{i,p}^{\pi}$ by considering the predicate cover of G. To recall, the predicate cover, denoted by $\mathit{pred\_cover}(F)$, of a formula F over a set of predicates is the weakest conjunctive formula over the predicates that implies F. Let $\phi(F, \mathit{preds}, i, \pi)$ denote the boolean formula over boolean variables $b_{i,p}^{\pi}$ obtained after replacing each predicate p in $\mathit{pred\_cover}(F)$ by $b_{i,p}^{\pi}$. For example, if the predicate cover is $x \leq y \wedge y \leq m$, then this boolean function is $b_{i,x\leq y}^{\pi} \wedge b_{i,y\leq m}^{\pi}$. The verification condition above can now be encoded as the following boolean constraint ψ_δ over boolean indicator variables $b_{i,p}^{\pi_1}$.

$$\psi_\delta \quad = \quad \bigwedge_{i=1}^{k} \phi(G, Q(I^{\pi_1}), i, \pi_1) \tag{3.4}$$

Case 3 *(Path between two adjacent cut-points)* We now combine the key ideas that we used in the above two cases to handle this more general case. The verification condition in Eq. 3.1 has the following form (after expanding I^{π_1} as $\bigvee_{i=1}^{k} I_i^{\pi_1}$ and I^{π_2} as $\bigvee_{j=1}^{k} I_j^{\pi_2}$, where each $I_i^{\pi_1}$ and $I_j^{\pi_2}$ is a conjunction of some predicates from

$Q(I^{\pi_1})$ and $Q(I^{\pi_2})$, respectively).

$$\begin{aligned} &\left(\bigvee_{i=1}^{k} I_i^{\pi_1}\right) \Rightarrow \left(G \Rightarrow \bigvee_{j=1}^{k} I_j^{\pi_2}\right) \\ \text{or, equivalently, } &\bigwedge_{i=1}^{k}\left(I_i^{\pi_1} \Rightarrow \left(G \Rightarrow \bigvee_{j=1}^{k} I_j^{\pi_2}\right)\right) \end{aligned} \qquad (3.5)$$

Using the same argument as in Case 1, the above constraint can be rewritten as:

$$\bigwedge_{i=1}^{k} \bigwedge_{p_1,..,p_k \in Q(I^{\pi_2})} \left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow \left(I_i^{\pi_1} \Rightarrow (G \Rightarrow \bigvee_{j=1}^{k} p_j) \right) \right)$$

Now, using the argument as in Case 2, the verification condition above can be encoded as the following boolean constraint ψ_δ over boolean indicator variables $b_{i,p}^{\pi_1}$ and $b_{i,p}^{\pi_2}$:

$$\psi_\delta \quad = \quad \bigwedge_{i=1}^{k} \bigwedge_{p_1,..,p_k \in Q} \left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow \phi\left((G \Rightarrow \bigvee_{j=1}^{k} p_j), Q(I^{\pi_1}), i, \pi_1 \right) \right) \quad (3.6)$$

The desired boolean formula $\psi_{\texttt{Prog}}$ is now given by the conjunction of formulas ψ_δ for all simple paths δ in the program.

Observe that the constraints are generated locally from the verification condition of each simple path. Hence, the satisfiability-based technique has the potential for efficient incremental verification, i.e., verification of a modified version of an already verified program, with support of an incremental SAT solver.

The next section describes a generalization of the reduction here to work over templates with arbitrary boolean structure, as opposed to just DNF, and will therefore use `OptimalNegativeSolutions` as opposed to predicate cover as we did here.

3.6.2 SAT Encoding for General Templates

For every unknown variable v and any predicate $q \in Q(v)$, we introduce a boolean variable b_q^v to denote whether the predicate q is present in the solution for v. We show how to encode the verification condition of the program `Prog` using a boolean formula $\psi_{\texttt{Prog}}$ over the boolean variables b_q^v. The boolean formula $\psi_{\texttt{Prog}}$ is constructed by making calls to `OptimalNegativeSolutions`, which is our theorem proving interface, and the constructed formula has the property that if it is satisfiable if and only if the program has invariants that are instantiations of the template using the predicate map Q (as we show in Theorem 3.2).

Notation Given a mapping $\{v_i \mapsto Q_i\}_i$ (where $Q_i \subseteq Q(v_i)$), let $\texttt{BC}(\{v_i \mapsto Q_i\}_i)$ denote the boolean formula that constrains the unknown variable v_i to contain all predicates from Q_i.

$$\texttt{BC}(\{v_i \mapsto Q_i\}_i) = \bigwedge_{i, q \in Q_i} b_q^{v_i}$$

3.6.2.1 Encoding VCs as SAT using `OptimalNegativeSolutions`

We first show how to generate the boolean constraint $\psi_{\delta,\tau_1,\tau_2}$ that encodes the verification condition corresponding to any tuple $(\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$. Let τ_2' be the template that is obtained from τ_2 as follows. If τ_2 is different from τ_1, then τ_2' is same as τ_2, otherwise τ_2' is obtained from τ_2 by renaming all the unknown variables to fresh unknown variables with `orig` denoting the reverse mapping that maps the fresh unknown variables back to the original. This renaming is important to ensure that each occurrence of an unknown variable in the formula $\texttt{VC}(\langle \tau_1, \delta, \tau_2' \rangle)$ is unique.

Note that each occurrence of an unknown variable in the formula $\texttt{VC}(\langle\tau_1, \delta, \tau_2\rangle)$ may not be unique when τ_1 and τ_2 refer to the same template, which is the case when the path δ goes around a loop.

A simple approach would be to use `OptimalSolutions` to compute all valid solutions for $\texttt{VC}(\langle\tau_1, \delta, \tau_2'\rangle)$ and encode their disjunction. But because both τ_1 and τ_2' are uninstantiated unknowns, the number of optimal solutions explodes. We describe below an efficient construction that involves invoking `OptimalNegativeSolutions` only over formulae with a smaller number of unknowns (the negative) for a small choice of predicates for the positive variables. The reduction is a generalization of the construction presented in the previous section.

Let $\rho_1, .., \rho_a$ be the set of positive variables and let $\eta_1, .., \eta_b$ be the set of negative variables in $\texttt{VC}(\langle\tau_1, \delta, \tau_2'\rangle)$. Consider any positive variable ρ_i and any $q_j \in Q'(\rho_i)$, where Q' is the map that maps an unknown v that occurs in τ_1 to $Q(v)$ and an unknown v that occurs in τ_2 to $Q(v)\sigma_t$. We require the predicate maps for the positive unknowns contain a predicate *true*. Consider the partial map $\sigma_{\{\rho_i, q_j\}_{i,j}}$ that maps ρ_i to $\{q_j\}$, i.e., maps all positive variables in the formula to some single predicate from their possible set. Let $S^{\{\rho_i, q_j\}_{i,j}}_{\delta,\tau_1,\tau_2}$ be the set of optimal solutions returned after invoking the procedure `OptimalNegativeSolutions` on the formula $\texttt{VC}(\langle\tau_1, \delta, \tau_2'\rangle)\sigma_{\{\rho_i, q_j\}_{i,j}}$ as below:

$$S^{\{\rho_i, q_j\}_{i,j}}_{\delta,\tau_1,\tau_2} = \texttt{OptimalNegativeSolutions}(\texttt{VC}(\langle\tau_1, \delta, \tau_2'\rangle)\sigma_{\{\rho_i, q_j\}_{i,j}}, Q')$$

The following Boolean formula $\psi_{\delta,\tau_1,\tau_2,\sigma_t}$ encodes the verification condition corresponding to $(\delta, \tau_1, \tau_2, \sigma_t)$.

$$\psi_{\delta,\tau_1,\tau_2,\sigma_t} = \bigwedge_{\rho_i, q_j \in Q'(\rho_i)} \left(\left(\bigwedge_{\rho_i} b^{\mathtt{orig}(\rho_i)}_{q_j \sigma_t^{-1}} \right) \Rightarrow \bigvee_{\{\eta_k \mapsto Q_k\}_k \in S^{\{\rho_i, q_j\}_{i,j}}_{\delta,\tau_1,\tau_2}} \mathtt{BC}(\{\mathtt{orig}(\eta_k) \mapsto Q_k \sigma_t^{-1}\}_k) \right) \tag{3.7}$$

This encoding makes use of the fact that there is an indicator variable for the empty set, corresponding to the predicate *true*, which is semantically identical to the empty set. Consequently, the antecedent will always be non-trivial.

The verification condition of the entire program is now given by the following boolean formula $\psi_{\mathtt{Prog}}$, which is the conjunction of the verification conditions of all tuples $(\delta, \tau_1, \tau_2, \sigma_t) \in \mathtt{Paths}(\mathtt{Prog})$.

$$\psi_{\mathtt{Prog}} \quad = \bigwedge_{(\delta,\tau_1,\tau_2,\sigma_t) \in \mathtt{Paths}(\mathtt{Prog})} \psi_{\delta,\tau_1,\tau_2,\sigma_t} \tag{3.8}$$

Example 3.10 *Consider the* `ArrayInit` *program from Example 3.3. Let* $Q(v) = Q_{j,\{0,i,n\}}$. *The above procedure leads to generation of the following constraints.*

Entry Case *The verification condition corresponding to this case contains one negative variable* v *and no positive variable. The set* S_{δ,τ_1,τ_2} *is same as the set* S *in Example 3.8, which contains* 4 *optimal solutions. The following boolean formula encodes this verification condition.*

$$(b^v_{0 \leq j} \wedge b^v_{j<i}) \vee (b^v_{0<j} \wedge b^v_{j \leq i}) \vee (b^v_{i \leq j} \wedge b^v_{j<0}) \vee (b^v_{i<j} \wedge b^v_{j \leq 0}) \tag{3.9}$$

Exit Case *The verification condition corresponding to this case contains one positive variable* v *and no negative variable. We now consider the set* $S^{v,q}_{\delta,\tau_1,\tau_2}$ *for each* $q \in Q(v)$. *Let* $P = \{0 \leq j, j < i, j \leq i, j < n, j \leq n\}$. *If* $v \in P$, *the set* $S^{v,q}_{\delta,\tau_1,\tau_2}$

contains the empty mapping (i.e., the resultant formula when v is replaced by q is valid). If $v \in Q(v) - P$, the set $S^{v,q}_{\delta,\tau_1,\tau_2}$ is the empty set (i.e., the resultant formula when v is replaced by q is not valid). The following boolean formula encodes this verification condition.

$$\bigwedge_{q \in P} (b^v_q \Rightarrow true) \wedge \bigwedge_{q \in Q(v)-P} (b^v_q \Rightarrow false)$$

which is equivalent to the following formula

$$\neg b^v_{0<j} \wedge \neg b^v_{i<j} \wedge \neg b^v_{i\leq j} \wedge \neg b^v_{n<j} \wedge \neg b^v_{n\leq j} \wedge \neg b^v_{j<0} \wedge \neg b^v_{j\leq 0} \tag{3.10}$$

Inductive Case *The verification condition corresponding to this case contains one positive variable v and one negative variable v' obtained by renaming one of the occurrences of v. Note that S_{δ,τ_1,τ_2} contains a singleton mapping that maps v' to the empty set. Also, note that $S^{v,j\leq i}_{\delta,\tau_1,\tau_2}$ is the empty set, and for any $q \in Q(v') - \{j \leq i\}$, $S^{v,q}_{\delta,\tau_1,\tau_2}$ contains at least one mapping that maps v' to the singleton $\{q\sigma_t\}$. Hence, the following boolean formula encodes this verification condition.*

$$(b^v_{j\leq i} \Rightarrow false) \;\wedge \bigwedge_{q \in Q(v')-\{j \leq i\}} \big(b^v_q \Rightarrow (b^v_q \vee \ldots)\big)$$

which is equivalent to the formula

$$\neg b^v_{j\leq i} \tag{3.11}$$

The boolean assignment where $b^v_{0\leq j}$ and $b^v_{j<i}$ are set to true, and all other boolean variables are set to $false$ satisfies the conjunction of the boolean constraints in Eq. 3.9,3.10, and 3.11. This implies the solution $\{0 \leq j, j < i\}$ for the unknown v in the invariant template.

The construction of the boolean constraint defined above satisfies the following property.

Theorem 3.2 *The boolean formula $\psi_{\texttt{Prog}}$ (Eq. 3.8) is satisfiable iff there exists an invariant solution for program* `Prog` *over predicate-map Q.*

In the interest of continuity, we present the proof of this theorem in Section A.4 (Appendix A.3).

3.7 Specification Inference

In this section, we address the problem of discovering *maximally weak* preconditions and *maximally strong* postconditions that fit a given template and ensure that all assertions in a program are valid.

3.7.1 Maximally Weak Pre- and Maximally Strong Postconditions

We first recap the definitions of maximally weak preconditions and maximally strong postconditions from the previous chapter by stating them formally.

Definition 3.4 (Maximally Weak Precondition) *Given a program* `Prog` *with assertions, invariant templates at each cutpoint, and a template $\tau_{\texttt{e}}$ at the program entry, we seek to infer a solution(s) σ to the unknowns in the templates such that*

- *σ is a valid solution, i.e.* $\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$.

```
GreatestFixedPointAll(Prog)
  Let $\sigma_0$ be s.t. $\sigma_0(v) \mapsto Q(v)$, if $v$ is negative
                         $\sigma_0(v) \mapsto \emptyset$, if $v$ is positive
  $S := \{\sigma_0\}$;
  while $S \neq \emptyset \wedge \exists \sigma \in S : \neg\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$
      Choose $\sigma \in S, (\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$ s.t.
                 $\neg\texttt{Valid}(\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t \rangle))\}$
      $S := S - \{\sigma\}$;
      Let $\sigma_p = \sigma \mid_{\texttt{Unk(Prog)}-\texttt{Unk}(\tau_1)}$.
      $S := S \cup \{\sigma' \cup \sigma_p \mid \bigwedge_{\sigma'' \in S} \tau_1\sigma' \not\Rightarrow \tau_1\sigma'' \ \wedge$
           $\sigma' \in \texttt{OptimalSolutions}(\texttt{VC}(\langle \tau_1, \delta, \tau_2\sigma\sigma_t \rangle), Q)$
  return $S$;
```

(a) Iterative Greatest Fixed-Point Computation

```
OptimallyWeakSolutions(Prog)
  $\phi := \phi_{\texttt{Prog}}$;
  $S := \emptyset$;
  while SAT($\phi$)
      $\phi' := \phi$;
      while SAT($\phi'$)
          $s :=$ SAT($\phi'$);
          $weak := (\tau_{\mathsf{e}}\ s \Rightarrow \tau_{\mathsf{e}}) \wedge \neg(\tau_{\mathsf{e}} \Rightarrow \tau_{\mathsf{e}}\ s)$;
          $\phi' := \phi \wedge \texttt{Boolify}(weak)$
      $S := S \cup \{s\}$;
      $\phi := \phi \wedge \neg\texttt{Boolify}(\tau_{\mathsf{e}} \Rightarrow \tau_{\mathsf{e}}\ s)$
  return $S$;
```

(b) Satisfiability-based Weakest Precondition Inference

Figure 3.10: Weakest precondition inference algorithms (a) using an iterative approach (described in terms of the procedure `OptimalSolutions`) (b) using a satisfiability-based approach that iteratively generates an increasingly weaker solution from a starting candidate.

```
LeastFixedPointAll(Prog, $Q$)
  Let $\sigma_0$ be s.t. $\sigma_0(v) \mapsto \emptyset$, if $v$ is negative
                         $\sigma_0(v) \mapsto Q(v)$, if $v$ is positive
  $S := \{\sigma_0\}$;
  while $S \neq \emptyset \wedge \exists \sigma \in S : \neg$Valid(VC(Prog, $\sigma$))
      Choose $\sigma \in S, (\delta, \tau_1, \tau_2, \sigma_t) \in$ Paths(Prog) s.t.
                         $\neg$Valid(VC($\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t \rangle$))
      $S := S - \{\sigma\}$;
      Let $\sigma_p = \sigma \mid_{\texttt{Unk(Prog)}-\texttt{Unk}(\tau_2)}$.
      $S := S \cup \{\sigma'\sigma_t^{-1} \cup \sigma_p \mid \bigwedge_{\sigma'' \in S} \tau_2\sigma'' \not\Rightarrow \tau_2\sigma'\sigma_t^{-1} \wedge$
          $\sigma' \in$ OptimalSolutions(VC($\langle \tau_1\sigma, \delta, \tau_2 \rangle$), $Q\sigma_t$)$\}$
  return $S$;
```

(a) Iterative Least Fixed-Point Computation

```
OptimallyStrongSolutions(Prog)
  $\phi := \phi_{\texttt{Prog}}$;
  $S := \emptyset$;
  while SAT($\phi$)
      $\phi' := \phi$;
      while SAT($\phi'$)
          $s :=$ SAT($\phi'$);
          $strong := (\tau_{\texttt{e}} \Rightarrow \tau_{\texttt{e}}\ s) \wedge \neg(\tau_{\texttt{e}}\ s \Rightarrow \tau_{\texttt{e}})$
          $\phi' := \phi \wedge$ Boolify($strong$)
      $S := S \cup \{s\}$;
      $\phi := \phi \wedge \neg$Boolify($\tau_{\texttt{e}}\ s \Rightarrow \tau_{\texttt{e}}$)
  return $S$;
```

(b) Satisfiability-based Strongest Postcondition Inference

Figure 3.11: Strongest postcondition inference algorithms (a) using an iterative approach (described in terms of the procedure OptimalSolutions) (b) using a satisfiability-based approach that iteratively generates an increasingly stronger solution from a starting candidate.

- *For any solution σ', it is not the case that $\tau_{\mathsf{e}}\sigma'$ is strictly weaker than $\tau_{\mathsf{e}}\sigma$, i.e.,*

$$\forall \sigma' : (\tau_{\mathsf{e}}\sigma \Rightarrow \tau_{\mathsf{e}}\sigma' \wedge \tau_{\mathsf{e}}\sigma' \not\Rightarrow \tau_{\mathsf{e}}\sigma) \Rightarrow \neg\mathtt{Valid}(\mathtt{VC}(\mathtt{Prog}, \sigma'))$$

Definition 3.5 (Maximally Strong Postcondition) *Given a program* `Prog`*, invariant templates at each cutpoint, and a template τ_{e} at program exit, we seek to infer a solution(s) σ to the unknowns in the templates such that*

- *σ is a valid solution, i.e.* $\mathtt{Valid}(\mathtt{VC}(\mathtt{Prog}, \sigma))$.
- *For any solution σ', it is not the case that $\tau_{\mathsf{e}}\sigma'$ is strictly stronger than $\tau_{\mathsf{e}}\sigma$, i.e.,*

$$\forall \sigma' : (\tau_{\mathsf{e}}\sigma' \Rightarrow \tau_{\mathsf{e}}\sigma \wedge \tau_{\mathsf{e}}\sigma \not\Rightarrow \tau_{\mathsf{e}}\sigma') \Rightarrow \neg\mathtt{Valid}(\mathtt{VC}(\mathtt{Prog}, \sigma'))$$

We now discuss how the iterative greatest and least fixed-point approaches can be extended to generate maximally weak preconditions and maximally strong postconditions, respectively.

Greatest fixed-points for maximally weak preconditions The greatest fixed-point based iterative technique described in Section 3.5.2 can be extended to generate maximally weak solutions as described in Figure 3.10(a). The only difference is that instead of generating only one maximally weak solution, we generate all maximally weak solutions (as is illustrated by the change in the while-loop condition in Figure 3.10(a) compared to that in Figure 3.8(b)).

Least fixed-points for maximally strong postconditions The least fixed-point based iterative technique described in Section 3.5.1 can be extended to generate maximally strong solutions as described in Figure 3.11(a). The only difference is that instead of generating only one maximally strong solution, we generate all maximally strong solutions (as is illustrated by the change in the while-loop condition in Figure 3.11(a) compared to that in Figure 3.8(a)).

The satisfiability-based approach can also be extended to compute solutions is based on a finite encoding that is similar to the approach for linear arithmetic (Section 2.4).

Satisfiability-based technique for maximally weak pre- and maximally strong postconditions The satisfiability-based technique described in Section 3.6 can be extended to generate maximally weak and maximally strong solutions as described in Figure 3.10(b) and Figure 3.11(b), respectively. The key idea is to first generate a boolean formula ϕ that encodes the verification condition of the program (Line 1) with the additional constraint that ϕ is not stronger than any of the maximally weak solutions already found (Line 10); or not weaker than any of the maximally strong solutions already found, respectively. Then, we construct a boolean formula ϕ' that encodes the additional constraint that the precondition τ_{e} should be strictly weaker or stronger than τ_{e} s (Line 8), where s is the last satisfying solution. If the formula ϕ' is satisfiable, we update s to the new satisfying solution (Line 6). We repeat this process in the inner loop (Lines 5-8) until the satisfying assignment s can be made weaker (for maximally weak precondition inference) and can be made stronger (for maximally strong postcondition inference).

3.8 Evaluation

We built a tool, called $\texttt{VS}^3_{\texttt{PA}}$, that implements the algorithms described in this chapter. We used the tool to verify and infer properties of various difficult benchmarks in our experiments.

We ran our experiments on a 2.5GHz Intel Core 2 Duo machine with 4GB of memory. We evaluated the performance of our algorithms over two sets of benchmark analyses. The first set consists of analyses that have been previously considered using alternative techniques. This serves to compare our technique based on SMT solvers against more traditional approaches. The second set consists of analyses that have not been feasible before.

3.8.1 Templates and Predicates

$\texttt{VS}^3_{\texttt{PA}}$ takes as input a program and a global set of templates and predicates. The global template is associated with each loop header (cut-point) and the global set of predicates with each unknown in the templates. We use a global set to reduce annotation burden, possibly at the cost of efficiency. The tool could potentially find solutions faster if different predicate sets were used for each invariant location, but the additional annotation burden would have been too cumbersome. For each benchmark program, we supplied the tool with a set of templates, whose structure is very similar to the program assertions (usually containing one unquantified unknown and a few quantified unknowns, as in Figures 3.1, 3.2, 3.3, and 3.4) and a set of predicates consisting of inequality relations between relevant program and bound

Benchmark	Assertion proved
Consumer Producer	$\forall k : 0 \leq k < n \Rightarrow C[k] = P[k]$
Partition Array	$\forall k : 0 \leq k < j \Rightarrow B[k] \neq 0$ $\forall k : 0 \leq k < l \Rightarrow A[k] = 0$
List Init, Del, Insert	$\forall k : x \rightsquigarrow k \wedge k \neq \perp \Rightarrow k \rightarrow val = 0$

Table 3.2: The assertions proved for verifying simple array/list programs.

Benchmark	LFP	GFP	CFP	Previous
Consumer Producer	0.45	2.27	4.54	45.00 [154]
Partition Array	2.28	0.15	0.76	7.96 [154], 2.4 [31]
List Init	0.15	0.06	0.15	24.5 [137]
List Delete	0.10	0.03	0.19	20.5 [137]
List Insert	0.12	0.30	0.25	23.9 [137]

Table 3.3: Time taken for verification of data-sensitive array and list programs.

variables.

3.8.2 Verifying standard benchmarks

We consider small but complicated programs that manipulate unbounded data structures. These programs have been considered in state-of-the-art alternative techniques that infer data-sensitive properties of programs.

Simple array/list manipulation: We present the performance of our algorithms on small but difficult programs manipulating arrays and lists. These benchmarks were culled from papers on state-of-the-art alternative techniques for verification. Table 3.2 presents the assertions that are proved by our algorithm. By adding axiomatic support for reachability, we were able to verify simple list programs illustrating our extensibility. Table 3.3 presents the benchmark examples, the time in

Benchmark	Assertion proved
Selection Sort Bubble Sort (n^2)	$\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2]$
Insertion Sort Bubble Sort (flag)	$\forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k{+}1]$

Table 3.4: The assertions proving that sorting programs output sorted arrays.

seconds taken by each of our algorithm (least fixed-point, greatest fixed-point and satisfiability-based) and the time reported by previous techniques[2].

Consumer Producer [154] is a loop that non-deterministically writes (produces) a new value into buffer at the head or reads (consumes) a value at the tail; we verify that the values read by the consumer are exactly those that are written by the producer. Partition Array [31, 154] splits an array into two separate arrays, one containing the zero entries and the other the non-zero; we verify that the resulting arrays indeed contain zero and non-zero entries. List Init [137] initializes the *val* fields of a list to 0; we verify that every node reachable from the head has been initialized. List Delete [137] (respectively, List Insert [137]) assumes a properly initialized list and deletes (respectively inserts) a properly initialized node; we verify that the resulting lists still have *val* fields as 0.

[2] We present the running times for previous techniques with the caveat that these numbers are potentially incomparable because of the differences in experimental setups and because some techniques infer predicates, possibly using hints. However, these comparisons substantiate the *robustness* of our approach in being able to infer invariants for all benchmarks, which individually required specialized theories earlier.

	Time (s)			
Benchmark	LFP	GFP	CFP	Previous
Selection Sort	1.32	6.79	12.66	na[3]
Insertion Sort	14.16	2.90	6.82	5.38 [145][3]
Bubble Sort (n^2)	0.47	0.78	1.21	na
Bubble Sort (flag)	0.22	0.16	0.55	na
Quick Sort (inner)	0.43	4.28	1.10	42.2 [137]
Merge Sort (inner)	2.91	2.19	4.92	334.1 [137]

Table 3.5: Time in seconds to verify sortedness for sorting programs.

Sortedness property: We choose sorting for our benchmark comparisons because these are some of the hardest verification instances for array programs that have been attempted by previous techniques. We verify sortedness for all major sorting procedures. Table 3.4 presents the assertions that we proved for these procedures.

Table 3.5 presents the benchmark examples, the time taken in seconds by our algorithms (least fixed-point, greatest fixed-point and satisfiability-based) to verify that they indeed output a sorted array and previously reported timings. We evaluate over selection, insertion and bubble sort (one that iterates n^2 times irrespective of array contents, and one that maintains a flag indicating whether the inner loop swapped any element or not, and breaks if it did not). For quick sort and merge sort we consider their partitioning and merge steps, respectively.

We do not know of a single technique that can uniformly verify all sorting benchmarks as is possible here. In fact, the missing results indicate that previous techniques are not robust and are specialized to the reasoning required for particular programs. In contrast, our tool successfully verified all programs that we attempted. Also, on time, we outperform the current state-of-the-art.

[3] [137] and [154] present timing numbers for the *inner loops* that are incomparable to the numbers

3.8.3 Proving ∀∃, worst-case bounds, functional correctness

We now present analyses for which no previous techniques are known. We handle three new analyses: ∀∃ properties verifying that sorting programs *preserve* the input elements, generating maximally weak preconditions for *worst case upper bounds* and *functional correctness*.

There are two key features of our algorithms that facilitate new and expressive analyses. The first is the ability to handle templates with arbitrary quantification to allow ∀∃ reasoning. Using this we verify preservation properties of sorting algorithms. The second, and arguably the more important characteristic, is the generation of greatest and least fixed-point solutions. We generate worst case upper bounds and maximally weak preconditions for functional correctness. Our experiments have shown that a satisfiability-based approach to generating least and greatest fixed-points gets stuck in the iterative process of making a solution optimal (inner loop of the algorithm in Figure 3.10(b)). We therefore restrict the use of the satisfiability-based approach to verification problems with the understanding that for maximally weak precondition it results in a time out.

∀∃ *properties:* Under the assumption that the elements of the input array are distinct, we prove the sorting algorithms do not lose any elements of the input. The

for the entire sorting procedure that we report here. For the inner loops of selection sort and insertion sort, our algorithms run in time 0.34(LFP), 0.16(GFP), 0.37(CFP) for selection sort compared to 59.2 [137] and in time 0.51(LFP), 1.96(GFP), 1.04(CFP) for insertion sort compared to 35.9 [137] and 91.22 [154].

Benchmark	Assertion proved
Selection, Insertion, Bubble (n^2, flag), Quick (inner) Sort	$\forall y \exists x : 0 \leq y < n \Rightarrow \tilde{A}[y] = A[x] \wedge 0 \leq x < n$
Merge Sort (inner)	$\forall y \exists x : 0 \leq y < m \Rightarrow A[y] = C[x] \wedge 0 \leq x < t$ $\forall y \exists x : 0 \leq y < n \Rightarrow B[y] = C[x] \wedge 0 \leq x < t$

Table 3.6: The assertions proved for verifying that sorting programs preserve the elements of the input. $\tilde{A}$ is the array A at the entry to the program.

	Time (s)		
Benchmark	LFP	GFP	CFP
Selection Sort	22.69	17.02	timeout
Insertion Sort	2.62	94.42	19.66
Bubble Sort (n^2)	5.49	1.10	13.74
Bubble Sort (flag)	1.98	1.56	10.44
Quick Sort (inner)	1.89	4.36	1.83
Merge Sort (inner)	timeout	7.00	23.75

Table 3.7: Time in seconds to verify preservation ($\forall\exists$) for sorting programs.

proof requires discovering $\forall\exists$ invariants (Table 3.6). The running times are shown in Table 3.7. Except for two runs that timeout, all three algorithms efficiently verify all instances.

Worst-case upper bounds: We have already seen that the worst-case input for Selection Sort involves a non-trivial precondition that ensures that a swap occurs every time it is possible (line 7 of Figure 3.3). For Insertion Sort we assert that the copy operation in the inner loop is always executed. For the termination checking version of Bubble Sort we assert that after the inner loop concludes the swapped flag is always set. For the partitioning procedure in Quick Sort (that deterministically chooses the leftmost element as the pivot), we assert that the pivot ends up at the rightmost location. All of these assertions ensure the respective worst-case runs

Benchmark	Precondition inferred
Selection Sort	$\forall k : 0 \leq k < n{-}1 \Rightarrow A[n{-}1] < A[k]$ $\forall k_1, k_2 : 0 {\leq} k_1 {<} k_2 {<} n{-}1 \Rightarrow A[k_1] < A[k_2]$
Insertion Sort	$\forall k : 0 \leq k < n{-}1 \Rightarrow A[k] > A[k{+}1]$
Bubble Sort (flag)	$\forall k : 0 \leq k < n{-}1 \Rightarrow A[k] > A[k{+}1]$
Quick Sort (inner)	$\forall k_1, k_2 : 0 \leq k_1 < k_2 \leq n \Rightarrow A[k_1] \leq A[k_2]$

Table 3.8: The preconditions inferred by our algorithms for worst case upper bounds runs of sorting programs.

Benchmark	Time (s)
Selection Sort	16.62
Insertion Sort	39.59
Bubble Sort (n^2)	0.00
Bubble Sort (flag)	9.04
Quick Sort (inner)	1.68
Merge Sort (inner)	0.00

Table 3.9: Time in seconds to infer preconditions for worst-case upper bounds of sorting programs.

occur.

We generate the maximally weak preconditions for each of the sorting examples as shown in Table 3.8. Notice that the inner loop of merge sort and the n^2 version of bubble sort always perform the same number of writes, and therefore no assertions are present and the precondition is *true*. The time taken is shown in Table 3.9, and is reasonable for all instances.

Functional correctness: Often, procedures expect conditions to hold on the input for functional correctness. These can be met by initialization, or by just assuming facts at entry. We consider the synthesis of the maximally weak such conditions. Table 3.10 lists our programs, the interesting non-trivial preconditions (*pre*) we

Benchmark	Preconditions inferred under given postcondition
Partial Init	*pre:* (a) $m \leq n$ (b) $\forall k : n \leq k < m \Rightarrow A[k] = 0$ *post:* $\forall k : 0 \leq k < m \Rightarrow A[k] = 0$
Init Synthesis	*pre:* (a) $i = 1 \wedge max = 0$ (b) $i = 0$ *post:* $\forall k : 0 \leq k < n \Rightarrow A[max] \geq A[k]$
Binary Search	*pre:* $\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2]$ *post:* $\forall k : 0 \leq k < n \Rightarrow A[k] \neq e$
Merge	*pre:* $\forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k{+}1]$ $\forall k : 0 \leq k < m \Rightarrow B[k] \leq B[k{+}1]$ *post:* $\forall k : 0 \leq k < t \Rightarrow C[k] \leq C[k{+}1]$

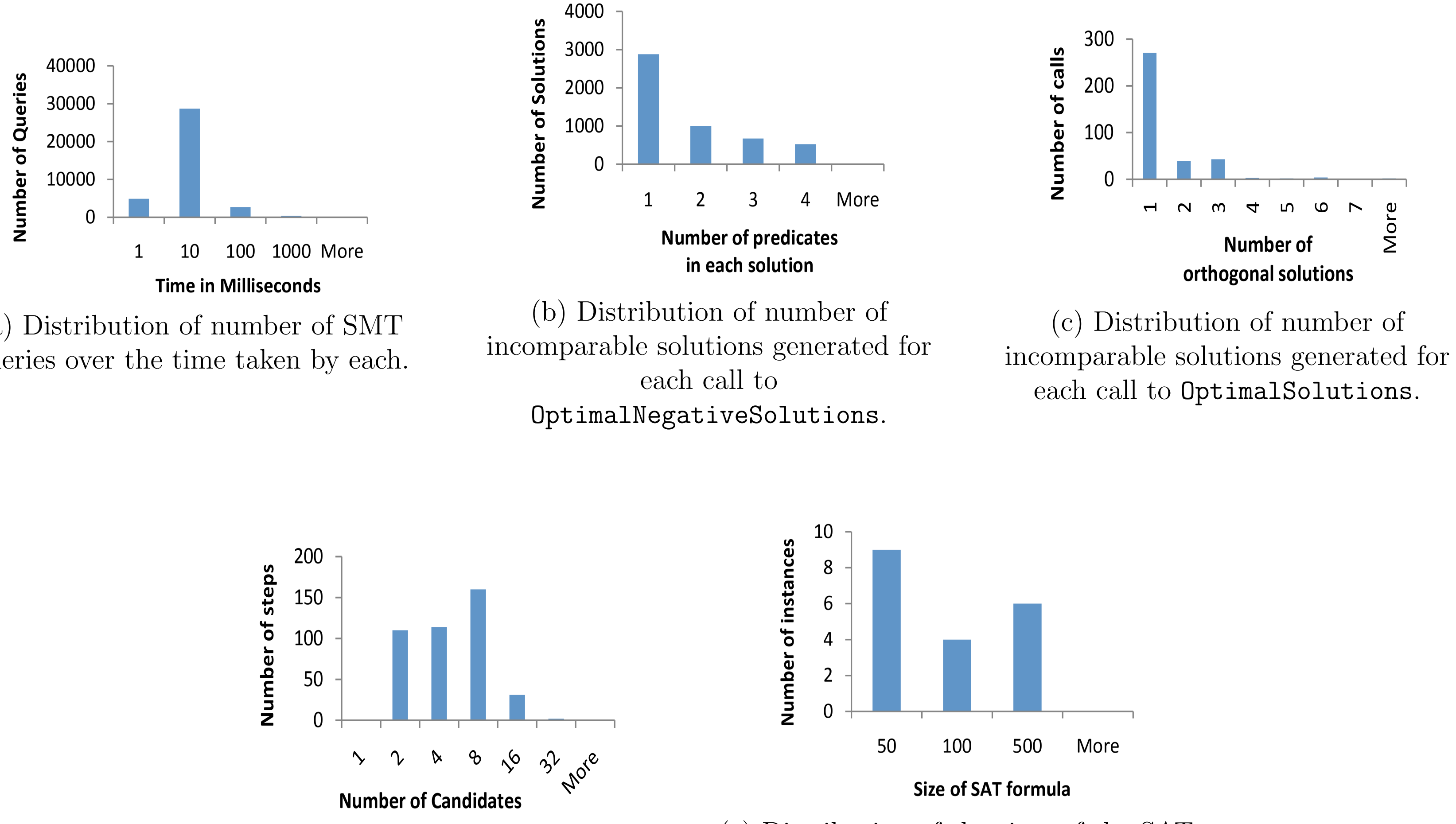
Table 3.10: Given a functional specification (post), the maximally weak preconditions (pre) inferred by our algorithms for functional correctness.

Benchmark	GFP
Partial Array Init	0.50
Init Synthesis	0.72
Binary Search	13.48
Merge Sort (inner)	3.37

Table 3.11: Time taken for maximally weak preconditions for functional correctness.

compute under the functional specification (*post*) supplied as postconditions. (We omit other non-interesting preconditions that do not give us more insights into the program but are generated by the tool nonetheless while enumerating maximally weak preconditions.) Table 3.11 lists the time taken to compute the preconditions.

Array Init initializes the locations $0 \ldots n$ while the functional specification expects initialization from $0 \ldots m$. Our algorithms, interestingly, generate two alternative preconditions, one that makes the specification expect less, while the other expects locations outside the range to be pre-initialized. Init Synthesis computes the index of the maximum array value. Restricting to equality predicates we compute two incomparable preconditions that correspond to the missing initializers. Notice that the second precondition is indeed maximally weak for the specification, even though max could be initialized out of bounds. If we expected to strictly output an array index and not just the location of the maximum, then the specification should have contained $0 \leq max < n$. Binary Search is the standard binary search for the element e with the correctness specification that if the element was not found in the array, then the array does not contain the element. We generate the precondition that the input array must have been sorted. Merge Sort (inner) outputs a sorted array. We infer that the input arrays must have been sorted for the procedure to be functionally correct.



(a) Distribution of number of SMT queries over the time taken by each.

(b) Distribution of number of incomparable solutions generated for each call to `OptimalNegativeSolutions`.

(c) Distribution of number of incomparable solutions generated for each call to `OptimalSolutions`.

(d) Distribution of the size of the candidate set across iterations.

(e) Distribution of the sizes of the SAT formula in terms of the number of clauses.

Figure 3.12: Statistical properties of our algorithms over predicate abstraction.

3.8.4 Properties of our algorithms

Statistical properties: We statistically examined the practical behavior our algorithms to explain why they work well despite the theoretical bottlenecks. We accumulated the statistics over all analyses and for all relevant modes (iterative and satisfiability-based).

First, we measured if the SMT queries generated by our system were efficiently decidable. Figure 3.12(a) shows that almost all of our queries take less than 10ms. By separating fixed-point computation from reasoning about local verification conditions, we have brought the theorem proving burden down to the realm of current solvers.

Second, because our algorithms rely on the procedures `OptimalSolutions` and `OptimalNegativeSolutions`, it is therefore important that in practice they return a small number of optimal solutions. In fact, we found that on most calls they return a single optimal solution (Figure 3.12(b) and 3.12(c)) and never more than 6. Therefore there are indeed a small number of possibilities to consider when they are called (on line 7 of Figures 3.6 and 3.8 and Eq. 3.7). This explains the efficiency of our local reasoning in computing the best abstract transformer.

Third, we examine the efficiency of the fixed-point computation (iterative) or encoding (satisfiability-based) built from the core procedures. For the iterative approaches, we reached a fixed-point in a median of 4 steps with the number of candidates remaining small, at around 8 (Figure 3.12(d)). This indicates that our algorithms perform a very directed search for the fixed-point. For the satisfiability-

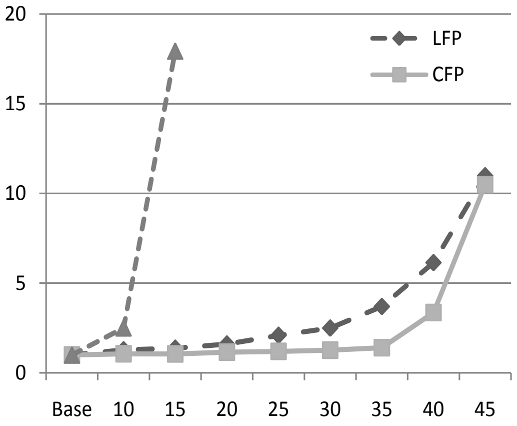


Figure 3.13: Robustness of invariant inference algorithms as we increase the number of redundant predicates. The x-axis denotes the extra predicates over the base set of predicates that prove the assertions, and the y-axis denotes the factor slowdown.

based approach, the number of clauses in the SAT formula never exceeds 500 (Figure 3.12(e)) with a median size of 5 variables. This explains the efficiency of our fixed-point computation.

Robustness: Our algorithms use a global set of user specified predicates. We evaluated the robustness of our algorithms over the sortedness analysis by adding irrelevant predicates. Figure 3.13 shows how the performance degrades, as a factor of the base performance and averaged over all sorting examples, as irrelevant predicates are introduced. The satisfiability-based approach is much more robust than the iterative schemes and, remarkably, only shows degradation past 35 irrelevant predicates. On the other hand, greatest fixed-point cannot handle more than 15 irrelevant predicates and least fixed-point shows steady decrease in performance with increasing number of irrelevant predicates.

3.8.5 Discussion

Our benchmark programs pose a spectrum of analysis challenges. The experiments corroborate the intuition that a universal panacea capable of addressing all these challenges probably does not exist. No single technique (forward or backward iterative, or bi-directional satisfiability-based) addresses all the challenges, but between them they cover the space of reasoning required. Therefore in practice, a combination will probably be required for handling real world instances.

We have also identified the different strengths that each algorithm demonstrates in practice. We found that for maximally weak precondition inference, the iterative greatest fixed-point approach is more efficient than the satisfiability-based approach. In a similar setting of computing maximally strong postcondition, the iterative least fixed-point is expected to be more efficient, as is indicated by its performance in our experiments. A satisfiability-based encoding is not suitable in an unconstrained problem where the number of possibilities grows uncontrollably. On the other hand, when the system is sufficiently constrained, for example when verifying sortedness or preservation, the satisfiability-based approach is significantly more robust to irrelevant predicates, followed by least fixed-point and lastly greatest fixed-point.

3.9 Summary

In this chapter, we have addressed the problem of inferring expressive program invariants over predicate abstraction for verification and also for inferring maximally weak preconditions. We presented the first technique that infers ∀ and ∀∃ quantified invariants for proving the full functional correctness of all major sorting algorithms. Additionally, we presented the first technique that infers maximally weak preconditions for worst-case upper bounds and for functional correctness.

We presented three fixed-point computing algorithms (two iterative and one satisfiability-based) that use a common basic interface to SMT solvers to construct invariants that are instantiations of templates with arbitrary quantification and boolean structure. Our algorithms can compute greatest and least fixed-point solutions that induce maximally weak precondition and maximally strong postcondition analyses.

We have implemented our algorithms in a tool that uses off-the-shelf SMT solvers. Our tool uniformly and efficiently verifies sortedness and preservation properties of all major sorting algorithms, and we have also used it for establishing worst-case bounds and maximally weak preconditions for functional correctness. We are unaware of any other technique that is able to perform these analyses.

3.10 Further Reading

Predicate abstraction Predicate abstraction was popularized by the model checking community, and in particular the BLAST [29, 148, 147] and SLAM [15, 14] model checkers. The success of these tools in automatically abstracting program states over a set of predicates (that could be arbitrarily complicated) allowed them to analyze complicated production C code [11, 13]. Subsequently, improvements such as symbolic predicate abstraction greatly improved the state-of-art in predicate abstraction-based model checking [177, 173].

Abstraction refinement An issue that we omit in this chapter is the construction of the abstraction, i.e., inferring the set of predicates to abstract over. A standard approach in the model checking community is to start with trivial approximations (e.g., with the single predicate *true*) and then iteratively refine it as verification fails. Each failed verification attempt yields a counterexample corresponding to which a refinement is constructed [10, 147, 133, 59]. It would be instructive to consider the application of these techniques to satisfiability-based invariant inference.

Use of templates for invariant inference The use of templates for restricting the space of invariants is not entirely new—although defining them as explicitly as we do here is. With the undecidability of program verification, such assumptions are to be expected. In fact, *domains* in abstract interpretation [72] are templates of sorts, just not as structured as we use in this dissertation. Abstraction refinement techniques have also used template to instantiate proof terms [147]. Lately, *refinement templates* have been used for inferring limited forms of dependent types [229].

Quantified invariants Quantification in invariants is critical for verifying important properties of programs. In fact, sorting programs are the staple benchmarks for the verification community precisely because they require complicated quantified invariants. Quantification imposes theoretical limitations in general, therefore we are limited to making our tools as robust as possible in practice. Previous approaches attempted to handle quantification at the analysis level, resulting in complicated decision procedures [137], or the full literal specification of quantified predicates [81], or use implicit quantification through free variables for limited properties [174, 176, 175, 112]. Our approaches area more robust for two reasons. First, we delegate the concern of reasoning about quantification to SMT solvers, which have been well engineered to handle quantified queries that arise in practice [85]. Thus as the handling of quantification gets more robust in these solvers, our tools will benefit. Even with the current technology, we found the handling of quantification robust for even the most difficult verification examples. Second, the queries generated by our system, through `OptimalNegativeSolutions`, which instantiates the templates with single predicates and uses `OptimalSolutions` to aggregate the information, are at the low end of the difficulty that current solvers can handle.

Axiomatization of reachability, transitive closure, types and further We model linked data structures using a simple axiomization of *reachability*. The reachability predicate $\rightsquigarrow(u, v)$, or the more readable infix $u \rightsquigarrow v$, relates heap locations u and v if v is reachable from u by following appropriate pointers [208] (or a ternary reachability predicate with a "between" element [172]). A typical axiom for reachability—parameterized by a function f that follows the appropriate pointer, e.g., the `next` field—is:

$$\forall : u \rightsquigarrow_f v \iff u = v \vee (f(u) \neq \bot \wedge f(u) \rightsquigarrow_f v)$$

A key technical detail is that first-order logic provers cannot handle transitivity required by reachability, because adding transitive closure to even simple decidable fragments of first order logic makes them undecidable [127, 151]. Therefore, suitable incomplete axiomatizations limit the scope of the predicates while being complete enough for most real programs [185, 151, 178, 53, 201].

Predicates have even been used to encode low-level types, e.g., using a `HasType` predicate [64], with appropriate axioms. This approach of defining an operator (e.g., `sel`, `upd`, $\rightsquigarrow$, `HasType`) and axioms stating its semantics generalizes beyond specific programming constructs and can be used for user-defined operators. For instance, in Chapter 4, we show how such an approach can define the semantics of examples such as Fibonacci and shortest path to verify or synthesize them. For Fibonacci, we define an operator `Fib` and its semantics using axioms:

$$\texttt{Fib}(0) = 0 \ \wedge \ \texttt{Fib}(1) = 1 \ \wedge \ \forall k : \texttt{Fib}(k) = \texttt{Fib}(k-1) + \texttt{Fib}(k-2)$$

We also imagine using such axiomatization for bottom-up modular reasoning and synthesis.

SMT Technology We briefly mention the basics of efficient backtracking algorithms for finding solutions to SAT and algorithms for combining these with decision procedures for solving SMT problems. The core backtracking algorithm, which is the basis of all modern SAT solvers, is the Davis-Putnam-Logemann-Loveland (DPLL) [84, 83] procedure. A basic backtracking process picks a literal and recursively checks if the two subproblems induced by assigning the literal *true* or $false$ are satisfiable. The solver outputs an assignment if the choices lead to the formula being satisfiable. Otherwise, it backtracks until all assignments have been explored and found unsatisfiable. DPLL adds two enhancements: (1) unit propagation, which checks for clauses with single literals and assigns the only satisfying choice to the literal, and (2) pure literal elimination, which checks for variables that occur only with one polarity (either negated or not) in the entire formula and assigns them such that their clauses are satisfied. Incredible engineering advances that work well in practice have been made to the original algorithm, such as two-watched literals, backjumping (non-clausal backtracking), conflict-driven lemma learning, and restarts. The reader is referred to literature [211, 169, 123] on this topic for detailed discussions.

SMT solvers extend the basic SAT solving engine by efficiently combining them with solvers $Solver_T$ for satellite theories T, using an efficient DPLL(T) procedure [211, 117]. DPLL(T) is more efficient than both the eager and lazy approaches to augmenting DPLL with theories. In the *eager* approach an equi-satisfiable SAT formula is constructed from the SMT formula, using a theory-specific translation to SAT, e.g., for equality with uninterpreted function (EUF) [48]. The eager approach requires such a translation for each theory, which may not exist. An alternative *lazy* approach assigns a propositional variable to all atoms in the SMT formula and generates a satisfying model for the resulting SAT. The model is then checked by the theory solvers and new clauses are added if the theory solvers find the boolean assignments to the atoms inconsistent. For instance, if the DPLL procedure generates a model with $x < y$ as $true$ and $x < y + 10$ as $false$, the linear arithmetic solver will find this model inconsistent. The lazy approach suffers from the inability of the theory solvers to direct the search—they only participate as validators.

The key to DPLL(T) is the way it overcomes the drawbacks of both the eager and the lazy approaches. Like the lazy approach, $Solver_T$ validates the choices made by the DPLL core, but additionally, it propagates literals of the SAT formula that are consequences in the theory T back to the SAT solver, thus guiding the search like the eager approach.

Chapter 4

Proof-theoretic Synthesis: Verification-inspired Program Synthesis

> *"Get the habit of analysis—analysis will in time enable synthesis to become your habit of mind."*
>
> — Frank Lloyd Wright[1]

This chapter describes a novel technique for the synthesis of imperative programs. Automated program synthesis has the potential to make the programming and design of systems easier by allowing the programs to be specified at a higher-level than executable code. In our approach, which we call proof-theoretic synthesis, the user provides an input-output functional specification, a description of the atomic operations in the programming language, and resource constraints. Our technique synthesizes a program, if there exists one, that meets the input-output specification and uses only the given resources.

The insight behind our approach is to interpret program synthesis as generalized program verification, which allows us to bring verification tools and techniques,

[1] American Architect and Writer, the most abundantly creative genius of American architecture. His Prairie style became the basis of 20th century residential design in the United States, 1867-1959.

such as those described in Chapters 2 and 3 to program synthesis. Our synthesis algorithm works by creating a program with unknown statements, unknown guards, unknown inductive invariants (proof certificate for safety), and unknown ranking functions (proof certificate for termination). It then generates constraints that relate the unknowns, which we show can be solved using existing verifiers.

We demonstrate the feasibility of the proposed approach by synthesizing programs in three different domains: arithmetic, sorting, and dynamic programming. Using verification tools from previous chapters, we are able to synthesize programs for complicated arithmetic algorithms including Strassen's matrix multiplication and Bresenham's line drawing; several sorting algorithms; and several dynamic programming algorithms. For these programs, the median time for synthesis is 14 seconds, and the ratio of synthesis to verification time ranges between 1× to 92× (with an median of 7×).

4.1 Program Synthesis as Generalized Verification

Automated program synthesis, despite holding the promise for significantly easing the task of programming, has received little attention due to its difficulty. Being able to mechanically construct programs has wide-ranging implications. Mechanical synthesis yields programs that are correct-by-construction. It relieves the tedium and error associated with programming low-level details, can aid in automated debugging, and in general leaves the human programmer free to deal with the high-level design of the system. Additionally, synthesis could discover new non-trivial programs that are difficult for programmers to build.

In this chapter, we present an approach to program synthesis that takes the correct-by-construction philosophy of program design [92, 130, 267] and shows how it can be automated. In the previous chapters, we described verification tools that can infer inductive invariants for partial correctness and ranking functions for termination. They do this by solving a system of implications (verification condition), with unknown invariants. In this chapter we show that it is possible to treat synthesis as a verification problem by encoding program guards and statements as additional logical facts that we trick the verifier into discovering—enabling use of existing verification tools for synthesis. The verification tool infers the invariants and ranking functions as usual, but in addition infers the program statements, yielding automated program synthesis. We call our approach *proof-theoretic synthesis* because the proof is synthesized alongside the program.

We use a novel definition of the synthesis task as requirements on the output program: functional requirements, requirements on the form of program expressions and guards, and requirements on the resources used (Section 4.2). The key to our synthesis algorithm is to treat synthesis as generalized verification by defining a reduction from the synthesis task to three sets of constraints. The first set are safety conditions that ensure the partial correctness of the loops in the program. The second set are well-formedness conditions on the program guards and statements, such that the output from the verification tool (facts corresponding to program guards and statements) correspond to valid guards and statements in an imperative language. The third set are progress conditions that ensure that the program terminates. We call these *synthesis conditions* and solve them using off-the-shelf verifiers (Section 4.3), such as the ones built in the previous chapters. We also present requirements that program verification tools must meet in order to be used for synthesis of program statements and guards (Section 4.4).

We build synthesizers using verifiers $\texttt{VS}^3_{\texttt{LIA}}$ and $\texttt{VS}^3_{\texttt{PA}}$ from previous chapters, and present synthesis results for the three domains of arithmetic, sorting and dynamic programming (Section 4.5). This approach not only synthesizes the program, but additionally the proof of correctness and termination alongside. To our knowledge, our approach is the first that automatically synthesizes programs and their proofs, while previous approaches have either used given proofs to extract programs [195] or not attempted to provide correctness guarantees at all [245].

4.1.1 Motivating Example: Bresenham's Line Drawing

To illustrate our approach, we next show how to synthesize Bresenham's line drawing algorithm. This example is ideal for automated synthesis because, while the program's requirements are simple to specify, the actual program is quite involved.

Bresenham's line drawing algorithm is shown in Figure 4.1(a). The algorithm computes (and writes to the output array *out*) the discrete best-fit line from $(0, 0)$ to (X, Y), where the point (X, Y) is in the NE half-quadrant, i.e., $0 < Y \leq X$. The best-fit line is one that does not deviate more than half a pixel away from the real line, i.e., $|y - (Y/X)x| \leq 1/2$. For efficiency, the algorithm computes the pixel values (x, y) of this best-fit line using only linear operations, but the computation is non-trivial and the correctness of the algorithm is also not evident.

An important idea underlying our approach is that we can write program statements as equality predicates, as we discussed in Chapter 3, and acyclic fragments as transition systems. We define transition systems formally in Section 4.3.1, and they essentially correspond to a set of guarded commands [88]. For example, we can write $x := e$ as $x' = e$, where x' is the output value of x. We will write statements as equalities between the output, primed, versions of the variables and the expression (over the unprimed versions of the variables). Also, guards that direct control flow in an imperative program can now be seen as guards for statement facts in a transition system. Figure 4.1(c) shows our example written in transition system form. To prove partial correctness, one can write down the inductive invariant for the loop and check that the verification condition for the program is in fact valid. The verification condition consists of four implications for the four paths corresponding to the entry, exit, and one each for the branches in the loop. Using standard verification condition generation, and writing the renamed version of invariant τ as τ', these are

$$\begin{array}{rcl}
(0 < Y \leq X) \wedge s_{\texttt{entry}} & \Rightarrow & \tau' \\
\tau \wedge \neg g_{\texttt{loop}} & \Rightarrow & \forall k : 0 \leq k \leq X \Rightarrow \\
 & & |2.out[k] - 2.(Y/X)k| \leq 1 \\
\tau \wedge g_{\texttt{loop}} \wedge g_{\texttt{body1}} \wedge s_{\texttt{body1}} & \Rightarrow & \tau' \\
\tau \wedge g_{\texttt{loop}} \wedge g_{\texttt{body2}} \wedge s_{\texttt{body2}} & \Rightarrow & \tau'
\end{array} \tag{4.1}$$

(a)

```
Bresenhams(int $X,Y$) {
  $v_1$:=$2Y-X$;$y$:=$0$;$x$:=$0$;
  while ($x \le X$)
  | $out$[$x$]:=$y$;
  | if ($v_1 < 0$)
  |   $v_1$:=$v_1$+$2Y$;
  | else
  |   $v_1$:=$v_1$+2($Y$-$X$);$y$++;
  | $x$++;
  return $out$;
}
```

(b)

Precondition:

$0 < Y \le X$

Postcondition:

$\forall k : 0 \le k \le X \Rightarrow |2.out[k] - 2.(Y/X)k| \le 1$

Invariant τ:

$$\left(\begin{array}{l} 0 < Y \le X \\ v_1 = 2(x+1)Y - (2y+1)X \\ 2(Y-X) \le v_1 \le 2Y \\ \forall k : 0 \le k < x \Rightarrow |2.out[k] - 2.(Y/X)k| \le 1 \end{array} \right)$$

Ranking function φ:

$X - x$

(c)

```
Bresenhams(int $X,Y$) {
  true $\rightarrow v_1' = 2Y - X \wedge y' = 0 \wedge x' = 0$
  while ($x \le X$)
  | $v_1 < 0 \rightarrow out'$=upd$(out,x,y) \wedge v_1' = v_1 + 2Y \wedge x' = x + 1$
  | $v_1 \ge 0 \rightarrow out'$=upd$(out,x,y) \wedge v_1' = v_1 + 2(Y–X) \wedge y' = y + 1 \wedge x' = x + 1$
  return $out$;
}
```

Figure 4.1: Motivating proof-theoretic synthesis. (a) Bresenham's line drawing algorithm (b) The invariant and ranking function that prove partial correctness and termination, respectively. (c) The algorithm written in transition system form, with statements as equality predicates, guarded appropriately (array writes are modeled using standard `upd` predicates).

where we use symbols for the various parts of the program:

$$\begin{aligned}
g_{\texttt{body1}} &: \; v_1 < 0 \\
g_{\texttt{body2}} &: \; v_1 \geq 0 \\
g_{\texttt{loop}} &: \; x \leq X \\
s_{\texttt{entry}} &: \; v_1' = 2Y - X \wedge y' = 0 \wedge x' = 0 \\
s_{\texttt{body1}} &: \; out' = \texttt{upd}(out, x, y) \wedge v_1' = v_1 + 2Y \wedge x' = x + 1 \\
s_{\texttt{body2}} &: \; out' = \texttt{upd}(out, x, y) \wedge v_1' = v_1 + 2(Y - X) \wedge y' = y + 1 \wedge x' = x + 1
\end{aligned} \tag{4.2}$$

As before we reason about arrays using McCarthy's select/update predicates [198], i.e., $out' = \texttt{upd}(out, x, y)$ corresponds to the assignment $out[x] := y$.

With a little bit of work, one can *validate* that the invariant τ shown in Figure 4.1(b) satisfies Eq. (4.1). Checking the validity of given invariants can be automated using SMT solvers [86]. In fact, powerful program verification tools such as $\texttt{VS}^3_{\texttt{LIA}}$ and $\texttt{VS}^3_{\texttt{PA}}$ can generate fixed-point solutions—inductive invariants such as τ—automatically. Aside from the satisfiability-based techniques we described in the previous chapters, other approaches such as constraint-based invariant generation [62], abstract interpretation [72], or model checking [58] can also be used for invariant inference.

The insight behind the technique in this chapter is to ask the question, if we can infer τ in Eq. (4.1), then is it possible to *infer* the guards g_i's or the statements s_i's at the same time? We have the found the answer to be yes, we can infer guards and statements as well, by suitably encoding programs as transition systems, asserting appropriate constraints, and then leveraging program verification techniques to do a systematic (lattice) search for unknowns in the constraints. Here the unknowns now represent both the invariants and the statements and guards. It turns out that a direct solution to the unknown guards and statements may be uninteresting, i.e., it may not correspond to real programs, so we need *well-formedness* constraints. Additionally, even if we synthesize valid programs, it may be that the programs are non-terminating, so we need *progress* constraints as well.

Suppose that the statements $s_{\texttt{entry}}$, $s_{\texttt{body1}}$, and $s_{\texttt{body2}}$, are unknown. A trivial satisfying solution to Eq. (4.1) may set all these unknowns to `false`. If we use a typical program verification tool that computes least fixed-points starting from $\perp$, then indeed, it will output this solution. On the other hand, let us make the conditional guards $g_{\texttt{body1}}$ and $g_{\texttt{body2}}$ unknown. Again, $g_{\texttt{body1}} = g_{\texttt{body2}} = \texttt{false}$ is a satisfying solution. We get uninteresting solutions because the unknowns are not constrained enough to ensure valid statements and control-flow. Statement blocks are modeled as $\bigwedge_i x'_i = e_i$ with one equality for each output variable x'_i and expressions e_i are over input variables. Therefore, `false` does not correspond to any valid block. Similarly $g_{\texttt{body1}} = g_{\texttt{body2}} = \texttt{false}$ does not correspond to any valid conditional with two branches. For example, consider `if` (g) S_1 `else` S_2 with two branches. Note how S_1 and S_2 are guarded by g and $\neg g$, respectively, and $g \vee \neg g$ holds. For every valid conditional, the disjunction of the guards is always a tautology. In verification, the program syntax and semantics ensure the *well-formedness* of acyclic fragments. In synthesis, we will need to explicitly constrain well-formedness of acyclic fragments (Section 4.3.4).

Next, suppose that the loop guard $g_{\texttt{loop}}$ is unknown. In this case if we attempt to solve for the unknowns τ and $g_{\texttt{loop}}$, then one valid solution assigns

$\tau = g_{\texttt{loop}} = \texttt{true}$, which corresponds to an non-terminating loop. In verification, we were only concerned with partial correctness and assumed that the program was terminating. In synthesis, we will need to explicitly *encode progress* by inferring appropriate ranking functions to prevent the synthesizer from generating non-terminating programs (Section 4.3.5).

Note that our aim is not to solve the completely general synthesis problem for a given *functional specification.* Guards and statements are unknowns but they take values from given domains, specified by the user as *domain constraints*, so that a lattice-theoretic search can be performed by existing program verification tools. Also notice that we did not attempt to change the number of invariants or the invariant position in the constraints. This means that we assume a given looping or *flowgraph structure*, e.g., one loop for our example. Lastly, as opposed to verification, the set of program variables is not known, and therefore we need a specification of the *stack space* available and also a bound on the type of *computations* allowed.

We use the specifications to construct an *expansion*, which is a program with unknown symbols and construct safety conditions over the unknowns. We then impose the additional well-formedness and progress constraints. We call the new constraints *synthesis conditions* and hope to find solutions to them using program verification tools such as $\texttt{VS}^3_{\texttt{LIA}}$ and $\texttt{VS}^3_{\texttt{PA}}$. The constraints generated are non-standard and therefore to solve them we need verification tools that satisfy certain properties. Our verification tools from the previous chapters do possess those properties. Indeed, satisfiability-based program verification tools can efficiently solve the synthesis conditions to synthesize programs (with a very acceptable slowdown over verification).

The guards, statements and proof terms for the example in this section come from the domain of arithmetic. Therefore, a verification tool for arithmetic such as $\texttt{VS}^3_{\texttt{LIA}}$ would be appropriate. For programs whose guards and statements are more easily expressed in other domains, a corresponding verification tool for that domain, such as $\texttt{VS}^3_{\texttt{PA}}$ for predicate abstraction, should be used. In fact, we have employed tools for the domains of arithmetic and predicate abstraction for proof-theoretic synthesis with great success. Our objective is to reuse existing verification technology—that started with invariant validation and progressed to invariant inference—and push it further to *program inference*.

4.2 The Synthesis Scaffold and Task

We now elaborate on the specifications that a proof-theoretic approach to synthesis requires and how these also allow the user to specify the space of interesting programs.

The following triple, called a *scaffold*, describes the synthesis problem:

$$\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$$

The components of this triple are:

1. Functional Specification The first component $\mathcal{F}$ of a scaffold describes the desired precondition and postcondition of the synthesized program. Let $\vec{v_{\texttt{in}}}$ and $\vec{v_{\texttt{out}}}$ be the vectors containing the input and output variables, respectively. Then a functional specification $\mathcal{F} = (F_{\texttt{pre}}(\vec{v_{\texttt{in}}}), F_{\texttt{post}}(\vec{v_{\texttt{in}}}, \vec{v_{\texttt{out}}}))$ is a tuple containing the formulae that hold at the entry and exit program locations. For example, for the program in Figure 4.1, $F_{\texttt{pre}}(X, Y) \doteq (0 < Y \leq X$ and $F_{\texttt{post}}(X, Y, out) \doteq \forall k : 0 \leq k \leq X \Rightarrow 2.(Y/X)k - 1 \leq 2.out[k] \leq 2.(Y/X)k + 1$.

2. Domain Constraints The second component $\mathcal{D} = (D_{\texttt{exp}}, D_{\texttt{grd}})$ of the scaffold describes the domains for expressions and guards in the synthesized program.

2a. *Program Expressions:* The expressions come from $D_{\texttt{exp}}$.

2b. *Program Guards:* The conditional and loop guards (boolean expressions) come from $D_{\texttt{grd}}$.

For example, for the program in Figure 4.1, the domains $D_{\texttt{exp}}$ and $D_{\texttt{grd}}$ are both linear arithmetic.

3. Resource Constraints The third component $\mathcal{R}$ of the scaffold describes the resources that the synthesized program can use. The resource specification $\mathcal{R} = (R_{\texttt{flow}}, R_{\texttt{stack}}, R_{\texttt{comp}})$ is a triple of resource templates that the user must specify for the flowgraph, stack and computation, respectively:

3a. *Flowgraph Template* We restrict attention to structured (or goto-less) programs, i.e., programs whose flowgraphs are reducible [146]. The structured nature of such flowgraphs allows us to describe them using simple strings. The user specifies $R_{\texttt{flow}}$ as a string from the following grammar:

$$T \ ::= \ \circ \ \mid \ *(T) \ \mid \ T;T$$

Here $\circ$ denotes an acyclic fragment of the flow graph, $*(T)$ denotes a loop containing the body T, and $T;T$ denotes the sequential composition of two flow graphs. For example, for the program in Figure 4.1, $R_{\texttt{flow}} = \circ;*(\circ)$.

3b. *Stack Template* The program is only allowed to manipulate a bounded number of variables, specified by means of a map $R_{\texttt{stack}} : \texttt{type} \rightarrow \texttt{int}$ indicating the number of extra temporary variables of each type. For example, for the program in Figure 4.1, $R_{\texttt{stack}} = (\texttt{int}, 1)$.

3c. *Computation Template* At times it may be important to put an upper bound on the number of times an operation is performed inside a procedure. A map $R_{\texttt{comp}} : \texttt{op} \rightarrow \texttt{int}$ of operations op to the upper bound specifies this constraint. For example, for the program in Figure 4.1, $R_{\texttt{comp}} = \emptyset$, which indicates that there are no constraints on computation.

While the resource templates make synthesis tractable by enabling a systematic lattice-theoretic search, they additionally allow the user to specify the space of interesting programs. While human programmers have a tendency to develop the simplest solutions, mechanical synthesizers do not. The resource templates formally enforce a suitability metric on the space of programs by allowing the user to restrict attention to desirable programs. For instance, the user may wish to reduce memory consumption at the expense of a more complex flowgraph and still meet the functional specification. If the user does not care, then the resource templates can be considered optional and left unspecified. In this case, the synthesizer can iteratively enumerate possibilities for each resource and attempt synthesis with increasing resources.

4.2.1 Picking a proof domain and a solver for the domain

Our synthesis approach is proof-theoretic, meaning we synthesize the proof terms, i.e., invariants and ranking functions, alongside the program. These proof terms will take values from a suitably chosen *proof domain* $D_{\texttt{prf}}$. Note that $D_{\texttt{prf}}$ must be at least as expressive as $D_{\texttt{grd}}$ and $D_{\texttt{exp}}$. The user chooses an appropriate proof domain and also picks a solver capable of handling that domain. We will use program verification tools, $\texttt{VS}^3_{\texttt{LIA}}$ and $\texttt{VS}^3_{\texttt{PA}}$, as solvers and typically, the user will pick the most powerful verification tool available for the chosen proof domain.

4.2.2 Synthesis Task

Given a scaffold $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$, we call an executable program *valid* with respect to the scaffold if it meets the following conditions.

- When called with inputs $\vec{v_{\texttt{in}}}$ that satisfy $F_{\texttt{pre}}(\vec{v_{\texttt{in}}})$ the program terminates, and the resulting outputs $\vec{v_{\texttt{out}}}$ satisfy $F_{\texttt{post}}(\vec{v_{\texttt{in}}}, \vec{v_{\texttt{out}}})$. There are associated invariants and ranking functions that provide a proof of this fact.
- There is a program loop (with an associated loop guard g) corresponding to each loop annotation (specified by "$*$") in the flowgraph template $R_{\texttt{flow}}$. The program contains statements from the following imperative language *IML* for

each acyclic fragment (specified by "$\circ$").

$$S \; ::= \; \texttt{skip} \;\mid\; S;S \;\mid\; x := e \;\mid\; \texttt{if } g \texttt{ then } S \texttt{ else } S$$

Where x denotes a variable and e denotes some expression. (Memory reads and writes are modeled using memory variables and select/update expressions.) The domain of expressions and guards is as specified by the scaffold, i.e., $e \in D_{\texttt{exp}}$ and $g \in D_{\texttt{grd}}$.

- The program uses only as many local variables as specified by $R_{\texttt{stack}}$ in addition to the input and output variables $\vec{v_{\texttt{in}}}, \vec{v_{\texttt{out}}}$.
- Each elementary operation only appears as many times as specified in $R_{\texttt{comp}}$.

Example 4.1 (Square Root) *Let us consider a scaffold with functional specification $\mathcal{F} = (x \geq 1, (i-1)^2 \leq x < i^2)$, which states that the program computes the integral square root of the input x , i.e., $i - 1 = \lfloor \sqrt{x} \rfloor$. Also, let the domain constraints $D_{\texttt{exp}}, D_{\texttt{grd}}$ be limited to linear arithmetic expressions, which means that the program cannot use any native square root or squaring operations. Lastly, let $R_{\texttt{flow}}$, $R_{\texttt{stack}}$ and $R_{\texttt{comp}}$ be $\circ;*(\circ);\circ$, $\{(\texttt{int}, 1)\}$ and $\emptyset$, respectively. A program that is valid with respect to this scaffold is the following:*

```
IntSqrt(int $x$) {
 $v$:=1;$i$:=1;
 while$^{\tau,\varphi}$($v \leq x$)
    $v$:=$v$+2$i$+1;$i$++;
 return $i-1$;
}
```

Invariant τ:

$v{=}i^2 \wedge x \geq (i{-}1)^2 \wedge i \geq 1$

Ranking function φ:

$x-(i{-}1)^2$

where v, i are the additional stack variable and loop iteration counter (and reused in the output), respectively. Also, the loop is annotated with the invariant τ and ranking function φ as shown, which prove partial correctness and termination, respectively.

We emphasize the notion of *validity* with respect to scaffolds of the synthesized programs:

Definition 4.1 (Validity with respect to a scaffold) *A terminating program P is* valid *with respect to a scaffold $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$, if it satisfies the Hoare triple $\{F_{\texttt{pre}}\}P\{F_{\texttt{post}}\}$, is in the language IML, has expressions and guards from the domains in $\mathcal{D}$, and uses only the resources as specified by $\mathcal{R}$.*

In the next two sections, we formally describe the steps of our synthesis algorithm. We first generate *synthesis conditions* (Section 4.3), which are constraints over unknowns for statements, guards, loop invariants and ranking functions. We then observe that they resemble verification conditions, and we can employ verification tools, if they have certain properties, to solve them (Section 4.4).

4.3 Synthesis Conditions

In this section, we define and construct *synthesis conditions* for an input scaffold $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$. Using the resource specification $\mathcal{R}$, we first generate a program with unknowns corresponding to the fragments we wish to synthesize. Synthesis conditions then specify constraints on these unknowns and ensure partial correctness, loop termination, and well-formedness of control-flow. We begin our discussion by motivating the representation we use for acyclic fragments in the synthesized program.

4.3.1 Using Transition Systems to Represent Acyclic Code

Suppose we want to infer a set of (straight-line) statements that transform a precondition $\phi_{\texttt{pre}}$ to a postcondition $\phi_{\texttt{post}}$, where the relevant program variables are x and y. One approach might be to generate statements that assigns unknown expressions e_x and e_y to x and y, respectively:

$$\{\phi_{\texttt{pre}}\}x := e_x; y := e_y\{\phi_{\texttt{post}}\}$$

Then we can use Hoare's axiom for assignment to generate the verification condition $\phi_{\texttt{pre}} \Rightarrow (\phi_{\texttt{post}}[y \mapsto e_y])[x \mapsto e_x]$. However, this verification condition is hard to automatically reason about because it contains substitution into unknowns. Even worse, we have restricted the search space by requiring the assignment to y to follow the assignment to x, and by specifying exactly two assignments.

Instead we will represent the computation as a transition system, which provides a much cleaner mechanism for reasoning when program statements are unknown. A *transition* in a transition system is a (possibly parallel) mapping of the input variables to the output variables. Variables have an input version and an output version (indicated by primed names), which allows them to change state. For our example, we can write a single transition:

$$\{\phi_{\texttt{pre}}\}\,\langle x', y'\rangle = \langle e_x, e_y\rangle\,\{\phi'_{\texttt{post}}\}$$

Here $\phi'_{\texttt{post}}$ is the postcondition, written in terms of the output variables, and e_x, e_y are expressions over the input variables. The verification condition corresponding to this tuple is $\phi_{\texttt{pre}} \wedge x' = e_x \wedge y' = e_y \Rightarrow \phi'_{\texttt{post}}$. Note that every state update (assignment) can always be written as a transition.

We can extend this approach to arbitrary acyclic program fragments. A *guarded transition* (written $[]g \rightarrow s$) contains a statement s that is executed only if the quantifier-free guard g holds. A *transition system* consists of a set $\{[]g_i \rightarrow s_i\}_i$ of guarded transitions. It is easy to see that a transition system can represent any arbitrary acyclic program fragment by suitably enumerating the paths through the acyclic fragment. The verification condition for $\{\phi_{\texttt{pre}}\}\{[]g_i \rightarrow s_i\}_i\{\phi'_{\texttt{post}}\}$ is simply $\bigwedge_i(\phi_{\texttt{pre}} \wedge g_i \wedge s_i \Rightarrow \phi'_{\texttt{post}})$.

In addition to the simplicity afforded by the lack of any ordering, the constraints from transition systems are attractive for synthesis as the program statements s_i and guards g_i are formulae just like the pre- and postconditions $\phi_{\texttt{pre}}$ and $\phi'_{\texttt{post}}$. Given the lack of differentiation, any (or all) can be unknowns in these *synthesis conditions*. This distinguishes them from verification conditions, which can at most have unknown invariants. Verification conditions are written with unknown invariants are used for invariant inference and with user-supplied invariants for invariant validation.

Synthesis conditions can thus be viewed as generalizations of verification conditions. Program verification tools routinely infer fixed-point solutions (invariants) that satisfy the verification conditions with known statements and guards. With our formulation of statements and guards as just additional facts in the constraints, it is possible to use sufficiently powerful verifiers such as $\mathtt{VS}^3_{\mathtt{LIA}}$ and $\mathtt{VS}^3_{\mathtt{PA}}$ to infer invariants *and* program statements and guards. *Synthesis conditions serve an analogous purpose to synthesis as verification conditions do to verification. If a program is correct (verifiable), then its verification condition is valid. Similarly, if a valid program exists for a scaffold, then its synthesis condition has a satisfying solution.*

4.3.2 Expanding a flowgraph

We synthesize code fragments for each acyclic fragment and loop annotation in the flowgraph template as follows:

- *Acyclic fragments* For each acyclic fragment annotation "$\circ$", we infer a transition system $\{g_i \to s_i\}_i$, i.e., a set of assignments s_i, stated as conjunctions of equality predicates, guarded by quantifier-free first-order-logic (FOL) guards g_i such that the disjunction of the guards is a tautology. Suitably constructed equality predicates and quantifier-free FOL guards are later translated to executable code—assignment statements and conditional guards, respectively—in the language *IML*.

- *Loops* For each loop annotation "$*$" we infer three elements. The first is the *inductive loop invariant* τ, which establishes partial correctness of each loop iteration. The second is the *ranking function* φ, which proves the termination of the loop. Both the invariant and ranking function take values from the proof domain, i.e., $\tau, \varphi \in D_{\mathtt{prf}}$. Third, we infer a quantifier-free FOL loop guard g.

Formally, the output of expanding flowgraphs will be a program in the transition system language *TSL* (note the correspondence to the flowgraph grammar):

$$p \ ::= \ \mathtt{choose}\ \{[]g_i \rightarrow s_i\}_i \quad | \quad \mathtt{while}^{\tau,\varphi}(g)\ \mathtt{do}\ \{p\} \quad | \quad p;p$$

Here each s_i is a conjunction of equality predicates, i.e., $\bigwedge_j \ (x_j = e_j)$. We will use $\vec{p}$ to denote a sequence of program statements in *TSL*. Note that we model memory read and updates using select/update predicates. Therefore, in $x = e$ the variable x could be a memory variable and e could be a memory select or update expression.

Given a string for a flowgraph template, we define an expansion function $\mathtt{Expand} : \mathtt{int} \times D_{\mathtt{prf}} \times \mathcal{R} \times \mathcal{D} \times R_{\mathtt{flow}} \rightarrow \textit{TSL}$ that introduces fresh unknowns for missing guards, statements and invariants that are to be synthesized. $\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(R_{\mathtt{flow}})$ expands a flowgraph $R_{\mathtt{flow}}$ and is parametrized by an integer n that indicates the number of transition each acyclic fragment will be expanded to, the proof domain, and the resource and domain constraints. The expansion outputs a program in the language *TSL*.

$$\begin{aligned}
\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(\circ) &= \mathtt{choose}\ \{[]g_i \rightarrow s_i\}_{i=1..n} \quad g_i, s_i \textit{ fresh unknowns} \\
\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(*(T)) &= \mathtt{while}^{\tau,\varphi}\ (g)\ \{ \qquad\qquad \tau, \varphi, g \textit{ fresh unknowns} \\
&\qquad\qquad \mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(T); \\
&\quad\ \} \\
\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(T_1;T_2) &= \mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(T_1);\mathtt{Expand}^{n,D_{\mathtt{prf}}}_{\mathcal{D},\mathcal{R}}(T_2)
\end{aligned}$$

Each unknown g, s, τ generated during the expansion has the following domain inclusion constraints.

$$\begin{aligned}
\tau &\in D_{\mathtt{prf}}|_V \\
g &\in D_{\mathtt{grd}}|_V \\
s &\in \textstyle\bigwedge_i x_i = e_i \quad \text{where } x_i \in V, e_i \in D_{\mathtt{exp}}|_V
\end{aligned}$$

Here $V = \vec{v_{\mathtt{in}}} \cup \vec{v_{\mathtt{out}}} \cup T \cup L$ is the set of variables: the input $\vec{v_{\mathtt{in}}}$ and output $\vec{v_{\mathtt{out}}}$ variables, the set of temporaries (local variables) T as specified by $R_{\mathtt{stack}}$, and the set of iteration counters and ranking function tracker variables is L (which we elaborate on later), one for each loop in the expansion. The restriction of the domains by the variable set V indicates that we are interested in the fragment of the domain over the variables in V. Also, the set of operations in e_i is bounded by $R_{\mathtt{comp}}$.

The expansion has some similarities to the notion of a user-specified *sketch* in previous approaches [247, 245]. However, the unknowns in the expansion here are more expressive than the integer unknowns considered in these prior approaches, and this allows us to perform a lattice search as opposed to the combinatorial approaches proposed earlier.

Example 4.2 *Let us revisit the integral square root computation from Example 4.1. Expanding the flowgraph template* ∘;∗(∘);∘ *with $n = 1$ yields exp_{sqrt}:*

```
choose {[]$g_1 \rightarrow s_1$} ;
while$^{\tau,\varphi}$ ($g_0$) {
   choose {[]$g_2 \rightarrow s_2$} ;
};
choose {[]$g_3 \rightarrow s_3$}
```

$$\begin{array}{rcl} \tau & \in & D_{\texttt{prf}}|_V \\ g_1, g_2, g_3 & \in & D_{\texttt{grd}}|_V \\ s_1, s_2, s_3 & \in & \bigwedge_i x_i = e_i \\ & & \quad x_i \in V, e_i \in D_{\texttt{exp}}|_V \end{array}$$

where $V = \{x, i, r, v\}$. The variables i and r are the loop iteration counter and ranking function tracker variable, respectively, and v is the additional local variable. Also, the chosen domains for proofs $D_{\texttt{prf}}$, guards $D_{\texttt{grd}}$, and expressions $D_{\texttt{exp}}$ are FOL facts over quadratic expressions, FOL facts over linear arithmetic, and linear arithmetic, respectively.

Notice that the expansion encodes everything specified by the domain and resource constraints and the chosen proof domain. The only remaining specification is $\mathcal{F}$, which we will use in the next section to construct safety conditions over the expanded scaffold.

4.3.3 Encoding Partial Correctness: Safety Conditions

Now that we have the expanded scaffold we need to collect the constraints (safety conditions) for partial correctness implied by the simple paths in the expansion. *Simple paths* (straight-line sequence of statements) start at a loop header $F_{\texttt{pre}}$ and end at a loop header or program exit. The loop headers, program entry, and program exit are annotated with invariants, precondition $F_{\texttt{pre}}$, and postcondition $F_{\texttt{post}}$, respectively.

Let ϕ denote formulae that represent pre- and postconditions and constraints. Then we define $\texttt{PathC} : \phi \times \mathit{TSL} \times \phi \rightarrow \phi$ as a function that takes a precondition, a sequence of statements, and a postcondition and outputs safety constraints that encode the validity of the Hoare triple. Let us first describe the simple cases of constraints from a single acyclic fragment and loop:

$$\begin{aligned}
&\texttt{PathC}(\phi_{\texttt{pre}}, (\texttt{choose}\ \{[]g_i \rightarrow s_i\}_i\ ), \phi_{\texttt{post}}) = \\
&\quad \textstyle\bigwedge_i (\phi_{\texttt{pre}} \wedge g_i \wedge s_i \Rightarrow \phi_{\texttt{post}}{}') \\
&\texttt{PathC}(\phi_{\texttt{pre}}, (\texttt{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}), \phi_{\texttt{post}}) = \\
&\quad \phi_{\texttt{pre}} \Rightarrow \tau' \wedge \texttt{PathC}(\tau \wedge g, \vec{p_l}, \tau) \wedge (\tau \wedge \neg g \Rightarrow \phi_{\texttt{post}}{}')
\end{aligned}$$

Here $\phi_{\texttt{post}}{}'$ and τ' are the postcondition $\phi_{\texttt{post}}$ and invariant τ but with all variables renamed to their output (primed) versions. Since the constraints need to refer to *output* postconditions and invariants the rule for a sequence of statements is a bit complicated. For simplicity of presentation, we assume that acyclic annotations do not appear in succession. This assumption holds without loss of generality because it is always possible to collapse consecutive acyclic fragments, e.g., two consecutive acyclic fragments with n transitions each can be collapsed into a single acyclic fragment with n^2 transitions. For efficiency, it is prudent not to make this assumption in practice, but the construction here generalizes easily. For a sequence of statements in *TSL*, under the above assumptions, there are three cases to consider. First, a loop followed by statements $\vec{p}$, whose reduction is as follows:

$$\begin{aligned}&\texttt{PathC}(\phi_{\texttt{pre}}, (\texttt{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\};\vec{p}), \phi_{\texttt{post}}) =\\ &\qquad (\phi_{\texttt{pre}} \Rightarrow \tau') \wedge \texttt{PathC}(\tau \wedge g, \vec{p_l}, \tau) \wedge \texttt{PathC}(\tau \wedge \neg g, \vec{p}, \phi_{\texttt{post}})\end{aligned}$$

Second, an acyclic fragment followed by just a loop, whose reduction is as follows:

$$\begin{aligned}&\texttt{PathC}(\phi_{\texttt{pre}}, (\texttt{choose}\ \{[]g_i \rightarrow s_i\}_i\ ;\texttt{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\}), \phi_{\texttt{post}}) =\\ &\qquad \textstyle\bigwedge_i(\phi_{\texttt{pre}} \wedge g_i \wedge s_i \Rightarrow \tau') \wedge \texttt{PathC}(\tau \wedge g, \vec{p_l}, \tau) \wedge (\tau \wedge \neg g \Rightarrow {\phi_{\texttt{post}}}')\end{aligned}$$

Third, an acyclic fragment, followed by a loop, followed by statements $\vec{p}$, whose reduction is as follows:

$$\begin{aligned}&\texttt{PathC}(\phi_{\texttt{pre}}, (\texttt{choose}\ \{[]g_i \rightarrow s_i\}_i\ ;\texttt{while}^{\tau,\varphi}\ (g)\ \{\vec{p_l}\};\vec{p}), \phi_{\texttt{post}}) =\\ &\qquad \textstyle\bigwedge_i(\phi_{\texttt{pre}} \wedge g_i \wedge s_i \Rightarrow \tau') \wedge \texttt{PathC}(\tau \wedge g, \vec{p_l}, \tau) \wedge \texttt{PathC}(\tau \wedge \neg g, \vec{p}, \phi_{\texttt{post}})\end{aligned}$$

The safety condition for a scaffold with functional specification $\mathcal{F} = (F_{\texttt{pre}}, F_{\texttt{post}})$, flowgraph template $R_{\texttt{flow}}$ and expansion $exp = \texttt{Expand}^{\mathcal{D},\mathcal{R}}_{n,D_{\texttt{prf}}}(R_{\texttt{flow}})$ is then given by:

$$\texttt{SafetyCond}(exp, \mathcal{F}) = \texttt{PathC}(F_{\texttt{pre}}, exp, F_{\texttt{post}}) \tag{4.3}$$

Example 4.3 *Consider the expanded scaffold (from Example 4.2) and the functional specification $\mathcal{F}$ (from Example 4.1) for integral square root. The loop divides the program into three simple paths, which results in* $\texttt{SafetyCond}(exp_{sqrt}, \mathcal{F})$*:*

$$\begin{array}{rcll} x \geq 1 \wedge g_1 \wedge s_1 & \Rightarrow & \tau' & \wedge \\ \tau \wedge g_0 \wedge g_2 \wedge s_2 & \Rightarrow & \tau' & \wedge \\ \tau \wedge \neg g_0 \wedge g_3 \wedge s_3 & \Rightarrow & (i'-1)^2 \leq x' \wedge x' < i'^2 & \end{array}$$

Notice that g_i, s_i, τ are all unknown placeholder symbols.

4.3.4 Encoding Valid Control: Well-formedness Conditions

We next construct constraints to ensure the well-formedness of `choose` statements. In the preceding development, we treated each path through the `choose` statement as independent. In any executable program control will always flow through at least one branch/transition of the statement, and each transition will contain well-formed assignment statements. We first describe a constraint that encodes this directly and then discuss an alternative way of ensuring well-formedness of transition guards.

Non-iterative upper bounded search We can parameterize the expansion of a scaffold by an integer n greater than the number of transitions expected in any acyclic fragment. The expanded scaffold can then represent any program that requires at most n-way branching in any acyclic fragment. Any excess transitions will have their guards instantiated to `false`. For any statement `choose` $\{[]g_i \rightarrow s_i\}$ in the expansion, we impose the well-formedness constraint:

$$\begin{aligned} \texttt{WellFormTS}(\{[]g_i \rightarrow s_i\}_i) \doteq & (\textstyle\bigwedge_i \texttt{valid}(s_i)) & \textit{Valid transition} \\ & \wedge (\textstyle\bigvee_i g_i) & \textit{Covers space} \end{aligned} \tag{4.4}$$

Here the predicate $\texttt{valid}(s_i)$ ensures *one and only one equality assignment* to each variable in s_i. This condition ensures that each s_i corresponds to a well-formed transition that can be translated to executable statements. The second term constrains the combination of the guards to be a tautology. Note that this is important to ensure that each transition system is well-formed and can be converted to a valid executable conditional. For example, consider the executable conditional `if` (G) `then` $x := E_1$ `else` $x := E_2$. The corresponding transition system is $\{[]g_1 \rightarrow (x' = E_1), []g_2 \rightarrow (x' = E_2)\}$, where $g_1 = G$ and $g_2 = \neg G$ and $g_1 \vee g_2$ holds. In *every* well-formed executable conditional the disjunction of the guards will be a tautology. This is that constraint imposed by the second term.

Notice that this construction does not constrain the guards to be disjoint. This is not required, as without loss of generality, the branches can be arbitrarily ordered (hence mutually exclusive) in the output to get a valid imperative program.

Iterative lower bounded search Notice that Eq. (4.4) is non-standard, i.e., it is not an implication constraint like typical verification conditions; and we will elaborate on this in Section 4.4. Program verification tools may or may not be able to handle such non-standard constraints. For example, the iterative approach from Chapter 3 cannot handle such non-standard constraints, while the satisfiability-based approaches from Chapters 2 and 3 can. Therefore, to enable use of a wider class verifiers, we discuss a technique for ensuring well-formedness of transitions without asserting Eq. (4.4).

We first assume that `valid`(s_i) holds, and we will show in Section 4.4.3 the conditions under which it does. Then all we need to ensure well-formedness is that $\vee_i g_i$ is a tautology. Since the transitions of a `choose` statement represent independent execution paths, we can perform an iterative search for the guards g_i. We start by finding *any* satisfying guard (and corresponding transition)—which can even be `false`. We then iteratively ask for another guard (and transition) such that the space defined by the new guard is *not* entirely contained in the space defined by the disjunction of the guards already generated. If we ensure that at each step the newly discovered guard covers some more space that was not covered by earlier guards, then eventually the disjunction of all will be a tautology.

More formally, suppose n such calls result in the transition system $\{[]g_i \rightarrow s_i\}_{i=1..n}$, and $\vee_{i=1..n} g_i$ is not already a tautology. Then for the $n+1^{st}$ transition, we assert the constraint $\neg(g_{n+1} \Rightarrow (\vee_{i=1..n} g_i))$. This constraint ensures that g_{n+1} will cover some space not covered by $\vee_{i=1..n} g_i$. We repeat until $\vee_i g_i$ holds. This iterative search for the transitions also eliminates the need to guess the value of n.

Well-formedness of an Expanded Scaffold We constrain the well-formedness of each transition system in the expanded scaffold $exp = \texttt{Expand}^{\mathcal{D},\mathcal{R}}_{n,D_{\texttt{prf}}}(R_{\texttt{flow}})$ using Eq. (4.4).

$$\texttt{WellFormCond}(exp) = \bigwedge_{\texttt{choose}\ \{[]g_i \rightarrow s_i\}_i\ \in \texttt{cond}(exp)} \texttt{WellFormTS}(\{[]g_i \rightarrow s_i\}_i) \tag{4.5}$$

where $\texttt{cond}(exp)$ recursively examines the expanded scaffold exp and returns the set of all `choose` statements in it.

Example 4.4 *For the expanded scaffold in Example 4.2, since each acyclic fragment only contains one guarded transition, the well-formedness constraints are simple and state that each of* $g_1, g_2, g_3 = \texttt{true}$ *and* $\texttt{valid}(s_1) \wedge \texttt{valid}(s_2) \wedge \texttt{valid}(s_3)$ *holds.*

4.3.5 Encoding Progress: Ranking functions

Until now our encoding has focused on safety conditions that, by themselves, only ensure partial correctness but not termination. Next, we add progress constraints to ensure that the synthesized programs terminate.

To encode progress for a loop $l = \texttt{while}^{\tau,\varphi_l}(g)\ \texttt{do}\ \{\vec{p}\}$, we assert the existence of a *ranking function* as an unknown (numerical) expression φ_l that is lower bounded and decreases with each iteration of the loop. Because φ_l is an *unknown expression* it is difficult to encode directly that it decreases. Therefore, we introduce a tracking variable r_l, such that $r_l = \varphi_l$. We use r_l to remember the value of the ranking function at the head of the loop, and because it is a proof variable no assignments to it can appear in the body of the loop. On the other hand, φ_l changes due to the loop body, and at the end of the iteration we can then check if the new value is strictly less than the old value, i.e., $r_l > \varphi_l$. Without loss of generality, we pick a lower bound of 0 for the tracking variable and conservatively assume that the termination argument is implied by the loop invariant τ, i.e, $\tau \Rightarrow r_l \geq 0$.

Now that we have asserted the lower bound, what remains is to assert that φ_l decreases in each iteration. Assume, for the time being, that the body does not contain any nested loops. Then we can capture the effect of the loop body using PathC as defined earlier, with precondition $\tau \wedge g$ and postcondition $r_l > \varphi$. Then, the progress constraint for loop l without any inner loop is:

$$\texttt{prog}(l) \quad \doteq \quad r_l = \varphi_l \wedge (\tau \Rightarrow r_l \geq 0) \wedge \texttt{PathC}(\tau \wedge g, \vec{p}, r_l > \varphi_l)$$

Using the above definition of progress we define the progress constraint for the entire expanded scaffold $exp = \texttt{Expand}^{\mathcal{D},\mathcal{R}}_{n,D_{\texttt{prf}}}(R_{\texttt{flow}})$:

$$\texttt{RankCond}(exp) = \bigwedge_{l \in \texttt{loops}(exp)} \texttt{prog}(l) \tag{4.6}$$

where $\texttt{loops}(exp)$ recursively examines the expanded scaffold exp and returns the set of all loops in it.

Example 4.5 *In the expanded scaffold of Example 4.2 there is only one loop, whose ranking function we denote by φ_l and with tracker r_l. Then we generate the following progress constraint:*

$$r_l = \varphi_l \wedge (\tau \Rightarrow r_l \geq 0) \wedge (\tau \wedge g_0 \wedge g_2 \wedge s_2 \Rightarrow r'_l > \varphi'_l)$$

To relax the assumption we made earlier about no nesting of loops, we need a simple modification to the progress constraint $\mathtt{prog}(l)$. Instead of considering the effect of the entire body $\vec{p}$ (which now contains inner loops), we instead consider the fragment $\mathtt{end}(l)$ *after* the last inner loop in $\vec{p}$. Also, let $\tau_{\mathtt{end}}$ denote the invariant for the last inner loop. Then, the progress constraint for loop l is:

$$\mathtt{prog}(l) \quad \doteq \quad r_l = \varphi_l \wedge (\tau \Rightarrow r_l \geq 0) \wedge \mathtt{PathC}(\tau_{\mathtt{end}}, \mathtt{end}(l), r_l > \varphi_l)$$

Notice that because the loop invariants are not decided a priori, i.e., we are *not* doing program extraction, we may assert that the invariants should be strong enough to satisfy the progress constraints. Specifically, we have imposed the requirement that the intermediate loop invariants carry enough information such that it suffices to consider only the last loop invariant $\tau_{\mathtt{end}}$ in the assertion.

4.3.6 Entire Synthesis Condition

Finally, we combine the constraints from the preceding sections to yield the entire synthesis condition for an expanded scaffold $exp = \mathtt{Expand}^{\mathcal{D},\mathcal{R}}_{n,D_{\mathtt{prf}}}(R_{\mathtt{flow}})$. The constraint $\mathtt{SafetyCond}(exp, \mathcal{F})$ (Eq. 4.3) ensures partial correctness of the program with respect to the functional specification. The constraint $\mathtt{WellFormCond}(exp)$

(Eq. 4.5) restricts the space to programs with valid control-flow. The constraint $\texttt{RankCond}(exp)$ (Eq. 4.6) restricts the space to terminating programs. The entire synthesis condition is given by

$$sc = \texttt{SafetyCond}(exp, \mathcal{F}) \wedge \texttt{WellFormCond}(exp) \wedge \texttt{RankCond}(exp)$$

Notice that we have omitted the implicit quantifiers for the sake of clarity. The actual form is $\exists U \forall V : sc$. The set V denotes the program variables, $\vec{v_{\texttt{in}}} \cup \vec{v_{\texttt{out}}} \cup T \cup L$ where T is the set of temporaries (additional local variables) as specified by the scaffold and L is the set of iteration counters and ranking function trackers. Also, U is the set of all unknowns of various types instantiated during the expansion of scaffold. This includes unknowns for the invariants τ, the guards g and the statements s.

Example 4.6 *Accumulating the partial correctness, well-formedness of branching and progress constraints we get the following synthesis condition (where we have removed the trivial guards g_1, g_2, g_3 as discussed in Example 4.4):*

$$\begin{array}{rcll}
x \geq 1 \wedge s_1 & \Rightarrow & \tau' & \wedge \\
\tau \wedge g_0 \wedge s_2 & \Rightarrow & \tau' & \wedge \\
\tau \wedge \neg g_0 \wedge s_3 & \Rightarrow & (i'-1)^2 \leq x' \wedge x' < i'^2 & \wedge \\
\multicolumn{3}{l}{\texttt{valid}(s_1) \wedge \texttt{valid}(s_2) \wedge \texttt{valid}(s_3)} & \wedge \\
\multicolumn{4}{l}{r_l = \varphi_l \wedge (\tau \Rightarrow r_l \geq 0) \wedge (\tau \wedge g_0 \wedge s_2 \Rightarrow r_l' > \varphi_l')}
\end{array}$$

```
Input: Scaffold $\langle \mathcal{F}, \mathcal{D}, \mathcal{R} \rangle$, maximum transitions $n$, proof domain $D_{\texttt{prf}}$
Output: Executable program or FAIL
begin
    $exp := \texttt{Expand}^{n, D_{\texttt{prf}}}_{\mathcal{D},\mathcal{R}}(R_{\texttt{flow}})$;
    $sc := \texttt{SafetyCond}(exp, \mathcal{F}) \wedge$
          $\texttt{WellFormCond}(exp) \wedge$
          $\texttt{RankCond}(exp)$;
    $\pi := \texttt{Solver}(sc)$;
    if $(\texttt{unsat}(\pi))$ then
        return FAIL;
    return $\texttt{Exe}^{\pi}(exp)$;
end
```

Figure 4.2: The proof-theoretic synthesis algorithm.

Here is a valid solution to the above constraints:

$$
\begin{aligned}
\tau &: \;\; v = i^2 \wedge x \geq (i-1)^2 \wedge i \geq 1 \\
g_0 &: \;\; v \leq x \\
\varphi_l &: \;\; x - (i-1)^2 \\
s_1 &: \;\; v' = 1 \wedge i' = 1 \wedge x' = x \wedge r_l' = r_l \\
s_2 &: \;\; v' = v + 2i + 1 \wedge i' = i + 1 \wedge x' = x \wedge r_l' = r_l \\
s_3 &: \;\; v' = v \wedge i' = i \wedge x' = x \wedge r_l' = r_l
\end{aligned}
\tag{4.7}
$$

Notice how each of the unknowns satisfy their domain constraints, i.e., τ is from FOL over quadratic relations, φ_l is a quadratic expression, s_1, s_2, and s_3 are conjunctions of linear equalities and g_0 is from quantifier-free FOL over linear relations. In the next section we show how such solutions can be computed using existing tools, e.g., the ones we developed in Chapters 2 and 3.

Under the assumption [90] that every loop with a pre- and postcondition has an inductive proof of correctness, and every terminating loop has a ranking function, and that the domains chosen are expressive enough, we can prove that the synthesis conditions, for the case of non-iterative upper bounded well-formedness, model the program faithfully:

Theorem 4.1 (Soundness and Completeness) *The synthesis conditions corresponding to a scaffold are satisfiable iff there exists a program (with a maximum of n transitions in each acyclic fragment where n is the parameter to the expansion) that is valid with respect to the scaffold.*

Additionally, for the alternative approach to discovering guards (Section 4.3.4), we can prove soundness and relative completeness:

Theorem 4.2 (Soundness and Relative Completeness) (a) Soundness: *If there exists a program that is valid with respect to the scaffold then at each step of the iteration the synthesis conditions generated are satisfiable.* (b) Relative completeness: *If the iterative search for guards terminates then it finds a program that is valid with respect to the scaffold.*

Corollary 4.1 (Completeness of Synthesis) *If there exists a program that is valid with respect to the scaffold, then the constraints generated are satisfiable and every satisfying solution corresponds to a program that is valid with respect to the scaffold.*

4.4 Solving Synthesis Conditions

In this section we describe how the synthesis conditions for an expanded scaffold can be solved using already existing fixed-point computation tools (program verifiers). We described two such tool, $\texttt{VS}^3_{\texttt{LIA}}$ and $\texttt{VS}^3_{\texttt{PA}}$ in the previous chapters. While our experiments were with these tools, we can employ any verifier, $\texttt{Solver}(sc)$, long as it meets certain requirements that we describe.

Suppose we have a procedure $\texttt{Solver}(sc)$ that can generate solutions to a synthesis condition sc. Figure 4.2 shows our synthesis algorithm, which expands the given scaffold to exp, constructs synthesis conditions sc, uses $\texttt{Solver}(sc)$ to generate a solution π to the unknowns that appear in the constraints, and finally generates concrete programs (whose acyclic fragments are from the language *IML* from Section 4.2) using the postprocessor $\texttt{Exe}^{\pi}(exp)$.

The concretization function $\texttt{Exe}^{\pi}(exp)$ takes the solution π that is computed by $\texttt{Solver}(sc)$ and the expanded scaffold exp, and outputs a program whose acyclic fragments are from the language *IML*. The function defines a concretization for each statement in *TSL* and annotates each loop with its loop invariant and ranking function:

$$
\begin{array}{rcl}
\texttt{Exe}^{\pi}(p;\vec{p}) & = & \texttt{Exe}^{\pi}(p);\texttt{Exe}^{\pi}(\vec{p}) \\
\texttt{Exe}^{\pi}(\texttt{while}^{\tau,\varphi}(g)\ \texttt{do}\ \{\vec{p}\}) & = & \texttt{while}^{\pi(\tau),\pi(\varphi_l)}(\pi(g))\ \{\ \texttt{Exe}^{\pi}(\vec{p})\ \} \\
\texttt{Exe}^{\pi}(\texttt{choose}\ \{[]g \rightarrow s\}) & = & \texttt{if}\ (\pi(g))\ \{\texttt{Stmt}(\pi(s))\}\ \texttt{else}\ \{\texttt{skip}\} \\
\texttt{Exe}^{\pi}(\texttt{choose}\ \{[]g_i \rightarrow s_i\}_{i=1..n}) & = & \qquad\qquad\qquad\qquad\qquad\qquad \text{(where } n > 1) \\
 & & \texttt{if}\ (\pi(g_1))\ \{\texttt{Stmt}(\pi(s_1))\} \\
 & & \texttt{else}\ \{\texttt{Exe}^{\pi}(\texttt{choose}\ \{[]g_i \rightarrow s_i\}_{i=2..n})\}
\end{array}
$$

where π maps each s to a conjunction of equalities and the concretization function $\texttt{Stmt}(s)$ expands the equality predicates to their corresponding state updates:

$$\texttt{Stmt}(\bigwedge_{i=1..n} x_i = e_i) \quad \doteq \quad (t_1 := e_1; \ldots; t_n := e_n);(x_1 := t_1; \ldots; x_n := t_n)$$

The above is a simple translation that uses additional fresh temporary variables $t_1 \ldots t_n$ to simulate parallel assignment. Alternatively, one can use data dependency analysis to generate code that uses fewer temporary variables.

4.4.1 Basic Requirement for $\texttt{Solver}(sc)$

Our objective is to use off-the-shelf verification tools to implement $\texttt{Solver}(sc)$, but we realize that not all tools are powerful enough. For use as a solver for synthesis conditions, verification tools require certain properties.

Let us first recall the notion of the polarity, *positive* or *negative*, of unknowns in a formula from Figure 3.5 in Chapter 3. Let ϕ be a FOL formula with unknowns whose occurrences are unique. Notice that all the constraints we generate have unique occurrences as we rename appropriately. An unknown is positive if strengthening it makes ϕ stronger. Analogously, an unknown is negative if weakening it makes the formula stronger. Also, recall that structurally, the nesting depth under negation defines whether an unknown is positive (even depth) or negative (odd depth). For example, the formula $(a \vee \neg b) \wedge \neg(\neg c \vee d)$ has positive unknowns $\{a, c\}$ and negative unknowns $\{b, d\}$.

In program verification we infer loop invariants given verification conditions with known program statements. Let us reconsider the verification condition in Eq. (4.1) with known program statements and guards. Notice that the implication constraints can be categorized into three forms; those with unknowns on both sides $\tau \wedge f_1 \Rightarrow \tau'$, those with unknowns only in the antecedent $\tau \wedge f_2 \Rightarrow f_3$, and those with unknowns only in the consequent $f_4 \Rightarrow \tau'$; where f_i's denote known formulae. Also, observe that these three are the only forms in which constraints in verification conditions can occur. From these, we can see that the verification conditions contain at most one positive and one negative unknown, depending on whether the corresponding path ends or starts at an invariant. Program verification tools implementing typical fixed-point computation algorithms are specialized to work solely with constraints with one positive and one negative unknown because there is no need to be more general.

In fact, traditional iterative fixed-point computation is even more specialized in that it requires support for either just one positive unknown or just one negative unknown. Traditional verifiers work either in a forward (computing least fixed-point) or backwards (computing greatest-fixed point) direction starting with the approximation $\bot$ or $\top$, respectively, and iteratively refining it.

A backwards iterative data flow analyzer always instantiates the positive unknown to the current approximation and uses the resulting constraint (with only one negative unknown) to improve the approximation. For example, suppose the current approximation to the invariant τ is f_5. Then a backwards analyzer may instantiate τ' in the constraint $\tau \wedge f_1 \Rightarrow \tau'$ to get the formula $\tau \wedge f_1 \Rightarrow f_5'$ (with one negative unknown τ). It will then use the formula to improve the approximation by computing a new value for τ that makes this formula satisfiable.

On the other hand, a typical forwards iterative data flow analyzer instantiates the negative unknown to the current approximation and uses the resulting constraint (with only one positive unknown) to improve the approximation. For example, suppose the current approximation to the invariant τ is f_6, then a forwards analyzer may instantiate τ in the constraint $\tau \wedge f_1 \Rightarrow \tau'$ to get the formula $f_6 \wedge f_1 \Rightarrow \tau'$ (with one positive unknown τ'). It will then use the formula to improve the approximation by computing a new value for τ' that makes this formula satisfiable.

In contrast, let us consider the components (from Section 4.3) of the synthesis condition. The component `SafetyCond`(exp) (Eq. (4.3)), in addition to the unknowns due to the invariants τ, contains unknowns for the program guards g and program statements s. These unknowns appear exclusively as negative unknowns, and there can be multiple such unknowns in each constraint. For example, in Eq. (4.1), the guards and statement unknowns appear as negative unknowns. On the other hand, the component `WellFormCond`(exp) (Eq. (4.5)) contains the well-formedness condition on the guards $\vee_i g_i$ that is a constraint with multiple positive unknowns. Therefore we need a verifier that satisfies the following.

Requirement 4.1 *Support for multiple positive and multiple negative unknowns.*

Notice this requirement is more general than that supported by typical verifiers we discussed above.

Now consider, an example safety constraint such as $\tau \wedge g \wedge s \Rightarrow \tau'$ with unknowns τ, g and s. This constraint can be rewritten as $\tau \Rightarrow \tau' \vee \neg g \vee \neg s$. Also, let us rewrite an example well-formedness constraint $\vee g_i$ as $\texttt{true} \Rightarrow \vee g_i$. This view presents an alternative explanation for Requirement 4.1 in that we need a tool that can infer the right case split, which in most cases would not be unique and would require maintaining multiple orthogonal solutions. Intuitively, this is related to a tool's ability to infer disjunctive facts.

In the above we implicitly assumed the invariant to be a conjunction of predicates. In the general case, we may wish to infer more expressive (disjunctive) invariants, e.g., of the form $u_1 \Rightarrow u_2$ or $\forall k : u_3 \Rightarrow u_4$, where u_i's are unknowns. In this case, multiple negative and positive unknowns appear even in the verification condition, and therefore the verification tool must satisfy Requirement 4.1, which matches the intuition that disjunctive inference is required.

4.4.2 Satisfiability-based Verifiers as $\texttt{Solver}(sc)$

Satisfiability-based fixed-point computation is a relatively recent approach to program verification that has been successfully used for difficult analyses. In previous chapters, we designed efficient satisfiability-based verification tools $\texttt{VS}^3_{\texttt{LIA}}$ and

$\texttt{VS}^3_{\texttt{PA}}$ for predicate abstraction (Chapter 3) and linear arithmetic (Chapter 2), respectively. Both $\texttt{VS}^3_{\texttt{LIA}}$ and $\texttt{VS}^3_{\texttt{PA}}$ satisfy Requirement 4.1.

Satisfiability-based verification tools reduce a verification condition vc (with invariant unknowns) to a boolean constraint $\psi(vc)$ such that a satisfying solution to the boolean constraint corresponds to valid invariants. Working with either linear arithmetic or predicate abstraction, the following is a restatement of results from previous chapters (Theorem 2.1 from Chapter 2, and Theorem 3.2 from Chapter 3) for satisfiability-based fixed-point computation:

Corollary 4.2 *The boolean constraint $\psi(vc)$ is satisfiable iff there exists a fixed-point solution for the unknowns corresponding to the invariants.*

The reduction can also be applied to synthesis condition sc to get boolean constraints $\psi(sc)$ and a similar property holds. The boolean constraint is satisfiable iff there exist satisfying statements, guards and invariants to the synthesis condition.

4.4.3 Iterative Verifiers as $\texttt{Solver}(sc)$

Let us now consider the case where the verification tool cannot handle non-standard constraints, such as Eq. (4.4). This is the case for typical iterative program verification tools that compute increasingly better approximations to invariants. We show that despite this lack of expressivity it is still possible to solve synthesis conditions as long as the tool satisfies an additional requirement.

The only constraint in the synthesis condition sc that is not an implication is $\texttt{WellFormCond}(sc)$. In Section 4.3.4, we discussed how an iterative lower-bounded search can discover the transitions $\{[]g_i \rightarrow s_i\}_i$ without asserting Eq. (4.5). There we had left the question of ensuring $\texttt{valid}(s_i)$ unanswered. Consider now the case where a valid solution g_i, s_i exists (i.e., s_i is not `false` or that $\texttt{valid}(s_i)$ holds) that satisfies the constraint set. As an instance, in Example 4.6, we have a synthesis condition for which a valid solution exists as shown by Eq. (4.7). Notice that this solution is strictly weaker than another solution that assigns identical values to other unknowns but assigns `false` to any of s_2, s_2, or s_3. In fact, we can observe that if the tool only generates maximally weak solutions then between these two solutions (which are comparable as we saw), it will always pick the one in which it does not assign `false` to statement unknowns. Therefore, it will always generate s_i such that $\texttt{valid}(s_i)$ holds unless no such s_i exists. As a result, if the program verification tool satisfies the following requirement, then we can omit Eq. (4.5) from the synthesis condition and still solve it using the tool.

Requirement 4.2 *Solutions are maximally weak.*

This requirement corresponds to the tool's ability to compute weakest preconditions. The typical approach to weakest preconditions (greatest fixed-point) computation propagates facts backwards, but this is considered difficult and therefore not many tools exist that do this. However, although traditional iterative data flow verifiers fail to meet Requirements 4.1 and 4.2, our iterative fixed-point computation approach from Chapter 3 computes maximally weak solutions and therefore satisfies the requirements.

In addition to ensuring $\texttt{valid}(s_i)$, maximally weak solutions also ensure that in each step of the iterative lower bounded search (Section 4.3.4), the algorithm will make maximal progress and converge faster. If the tool did not generate maximally weak solutions, then the iterative search for guards could take many more iterations to converge to a tautology. The downside is that the tool does more work than required. We require maximally weak solutions only for the statement unknowns, but instead the tool will generate maximally weak solutions for guards and invariants as well. This is not needed for synthesis as we are interested in *any* solution that satisfies the synthesis condition. Thus, the satisfiability-based scheme (which computes any fixed-point in the lattice rather than the greatest fixed-point) outperforms the iterative scheme in our experiments. In fact, tools based on iterative approximations do not terminate for most benchmarks, and we therefore perform the experiments using satisfiability-based tools.

4.5 Experimental Case Studies

To evaluate our approach, we synthesized examples in three categories: First, easy to specify but tricky to program *arithmetic* programs; second, *sorting* programs which all have the same specification but yield different sorting strategies depending on the resource constraints; third, *dynamic programming* programs for which naive solutions yield exponential runtimes, but which can be computed in polynomial time by suitable memoization.

4.5.1 Implementation

We implement our synthesis algorithm using existing satisfiability-based verifiers $\mathtt{VS}^3_{\mathtt{LIA}}$ and $\mathtt{VS}^3_{\mathtt{PA}}$, but which we augment as described below. Also, to simplify user input, we expanded user specified flowgraphs to be more expressive for certain cases.

Verification Tools Our synthesis technique relies on an underlying program verification tool. We took our $\mathtt{VS}^3_{\mathtt{LIA}}$ and $\mathtt{VS}^3_{\mathtt{PA}}$ verifiers and used them as synthesis solvers. These tools are state-of-the-art and can infer expressive invariants such as those requiring quantification and disjunction. However, for some of the benchmarks, the reasoning required was beyond even these their capabilities. We therefore extended the base verifiers with the following features.

- *Quadratic expressions for arithmetic* For handling quadratic expressions in the proofs, we implemented a sound but incomplete technique that renames quadratic expressions to fresh variables and then uses linear arithmetic reasoning of $\mathtt{VS}^3_{\mathtt{LIA}}$. We will discuss this encoding in detail in Section 6.2.2.2. This encoding suffices for most of our benchmarks except for one (integral square root), which we handle by explicitly encoding an assumption. We call this augmented solver $\mathtt{VS}^3_{\mathtt{QA}}$.

- *Axiomatization* Proposals exist for extending verification tools with axioms for theories they do not natively support, e.g., the theory of reachability for lists [172]. We take such axiomatization a step further and allow the user to specify axioms over uninterpreted symbols that define computations. We implement this in $\mathtt{VS}^3_{\mathtt{PA}}$ to specify the meaning of dynamic programming programs, e.g., the definition of Fibonacci. We call this augmented solver $\mathtt{VS}^3_{\mathtt{AX}}$.

Note that these extensions are to facilitate verification and not synthesis. The synthesis solver is exactly the same as the verification tool. The details of these extensions are presented in Chapter 6.

Flowgraphs with Init/Final Phases In practice a fair number of loops have characteristic *initialization* and *finalization* phases that exhibit behavior different from the rest of the loop. In theory, verifiers should be able to infer loop invariants that capture such semantically different phases. However, this requires disjunctive reasoning, which is fairly expensive if at all supported by the verifier. In particular, while our tools $\mathtt{VS}^3_{\mathtt{LIA}}$ and $\mathtt{VS}^3_{\mathtt{PA}}$ do support disjunctions, it is more expensive to handle than just conjunctive facts. On the other hand, other tools require non-trivial work to be lifted to disjunctive reasoning. For instance, abstract interpretation-based tools require expensive disjunction completions of domains [76, 118].

We use an alternate expansion $\overline{\mathtt{Expand}}^n(T)$ that introduces acyclic fragments for the initialization and finalization if synthesis without them fails. For instance, for Example 4.1, the the user only needs to specify the flowgraph $*(\circ)$ instead of the more complicated $\circ;*(\circ);\circ$. Except for the expansion of loops, $\overline{\mathtt{Expand}}^n(T)$ expands all other statements exactly like $\mathtt{Expand}^n(T)$ does. For loops, it builds an initialization and finalization phase as follows.

$$\overline{\texttt{Expand}}^n(*(T)){=}\texttt{Expand}^n(\circ); \qquad \rightarrow \textit{Added initialization}$$

$$\texttt{while}^\tau\ (g)\ \{\overline{\texttt{Expand}}^n(T);\}$$

$$\texttt{Expand}^n(\circ); \qquad \rightarrow \textit{Added finalization}$$

4.5.2 Algorithms that use arithmetic

For this category, we pick $D_{\texttt{prf}}$ to be quadratic arithmetic and use as our solver the $\texttt{VS}^3_{\texttt{QA}}$ tool. We chose a set of arithmetic benchmarks with simple-to-state functional specifications but each containing some tricky insight that human programmers may miss.

Swapping without Temporaries Consider a program that swaps two integer-valued variables *without* using a temporary. The precondition and postcondition to the program are specified as $F_{\texttt{post}} \doteq (x = c_2 \wedge y = c_1)$ and $F_{\texttt{pre}} \doteq (x = c_1 \wedge y = c_2)$, respectively. We specify an acyclic flowgraph template $R_{\texttt{flow}} \doteq \circ$ and a computation template $R_{\texttt{comp}} \doteq \emptyset$ that imposes no constraints. To ensure that no temporaries are used we specify $R_{\texttt{stack}} \doteq \emptyset$. The synthesizer generates various versions of the program, e.g.,

$$\texttt{Swap(int}\ x, y)\{x := x + y; y := x - y; x := x - y; \}$$

The synthesizer also finds numerous other alternative programs that are semantically equivalent, e.g.,

$$\texttt{Swap(int}\ x, y)\{x := x - y; y := x + y; x := -x + y; \}$$

Since we allow for non-trivial sized bit vectors for the coefficients, the total number of alternative solutions enumerated is of the order of thousands.

Strassen's 2×2 *Matrix Multiplication* Consider Strassen's matrix multiplication, which computes the product of two $n \times n$ matrices in $\Theta(n^{2.81})$ time instead of $\Theta(n^3)$. The key to this algorithm is an acyclic fragment that computes the product of two 2×2 input matrices $\{a_{ij}, b_{ij}\}_{i,j=1,2}$ using 7 multiplications instead of the expected 8. Used recursively, this results in asymptotic savings. The key insight of the algorithm lies in this core. Recursive block multiplication was well known, and Strassen augmented it with an efficient core. We synthesize the crucial acyclic fragment, which is shown in Figure 4.3. Here the precondition $F_{\texttt{pre}}$ is `true` and the postcondition $F_{\texttt{post}}$ is the conjunction of four equalities as (over the outputs $\{c_{ij}\}_{i,j=1,2}$):

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

The synthesizer also generates many alternate versions that are functionally equivalent to Figure 4.3.

As a side note, we also attempted synthesis using 6 multiplications, which failed. This suggests that possibly no asymptotically faster solution exists using simple quadratic computations—theoretical results up to $n^{2.376}$ are known [70], but use products that cannot be easily be captured in the simple domains considered here.

```
Strassens(int $a_{ij}, b_{ij}$) {
  $v_1$:=$(a_{11}+a_{22})(b_{11}+b_{22})$
  $v_2$:=$(a_{21}+a_{22})b_{11}$
  $v_3$:=$a_{11}(b_{12}-b_{22})$
  $v_4$:=$a_{22}(b_{21}-b_{11})$
  $v_5$:=$(a_{11}+a_{12})b_{22}$
  $v_6$:=$(a_{21}-a_{11})(b_{11}+b_{12})$
  $v_7$:=$(a_{12}-a_{22})(b_{21}+b_{22})$
  $c_{11}$:=$v_1+v_4-v_5+v_7$
  $c_{12}$:=$v_3+v_5$
  $c_{21}$:=$v_2+v_4$
  $c_{22}$:=$v_1+v_3-v_2+v_6$
  return $c_{ij}$;
}
```

Figure 4.3: Synthesis result for Strassen's Matrix Multiplication using the arithmetic solver.

Integral Square Root Consider computing the integral square root $\lfloor\sqrt{x}\rfloor$ of a positive number x using only linear or quadratic operations. The precondition is $F_{\texttt{pre}} \doteq x \geq 1$ and the postcondition, involving the output i, is $F_{\texttt{post}} \doteq (i-1)^2 \leq x < i^2$. We provide a single loop flowgraph template $R_{\texttt{flow}} \doteq *(\circ)$ and an empty computation template $R_{\texttt{comp}} \doteq \emptyset$. The synthesizer generates different programs depending on the domain constraints and the stack template:

- $R_{\texttt{stack}} \doteq \{(\texttt{int}, 0)\}$ and we allow quadratic expressions in $D_{\texttt{exp}}$ and $D_{\texttt{grd}}$. The synthesized program does a sequential search downwards starting from $i = x$ by continuously recomputing and checking $(i-1)^2$ against x.

- $R_{\texttt{stack}} \doteq \{(\texttt{int}, 1)\}$ and we only allow linear expressions in $D_{\texttt{exp}}$ and $D_{\texttt{grd}}$. The synthesized program does a sequential search but uses the additional local variable (rather surprisingly) to track the value of $(i-1)^2$ using only linear updates. The synthesized program is Example 4.1, from earlier.

- $R_{\texttt{stack}} \doteq \{(\texttt{int}, 2)\}$ and we allow quadratic expressions in $D_{\texttt{exp}}$ and $D_{\texttt{grd}}$. The synthesized program does a binary search for the value of i and uses the two additional local variables to hold the low and high end of the binary search space.

Notice that the stack template only specifies an upper bound. As such, for successively higher number of variables programs that use fewer variables are also valid solutions. The synthesizer generates all solutions, in particular, including those that use fewer variables than what the stack template specifies. We use the enumeration facility in satisfiability-based verifiers to enumerate all valid solutions. In the above description, for higher number of variables, we mention that programs that are generated *in addition* to the ones before.

Bresenham's Line Drawing Algorithm Consider Bresenham's line drawing algorithm, as we discussed in Section 4.1.1. For efficiency, the algorithm only uses linear updates, which are non-trivial to verify [107] or even understand (let alone discover from scratch).

We specify the precondition $F_{\texttt{pre}} \doteq 0 < Y \leq X$. The postcondition (as presented in Section 4.1.1) is quantified, but $\texttt{VS}^3_{\texttt{QA}}$ does not support quantification. Therefore we provide a facility to annotate the flowgraph template with the assertion $|2y - 2(Y/X)x| \leq 1$ at the loop header and specify that the loop iterates over $x = 0\,..\,X$. This indicates the tradeoffs we can make in our technique. The user can offset the limitations of the available verification tool by indicating extra known values in the scaffold. We specify a single loop flowgraph $R_{\texttt{flow}} \doteq *(\circ)$ and empty stack and computation templates $R_{\texttt{stack}} \doteq \emptyset$, $R_{\texttt{comp}} \doteq \emptyset$. The synthesizer generates multiple versions, one of which is shown in Figure 4.1(a).

4.5.3 Sorting Algorithms

For this category, we pick $D_{\texttt{prf}}$ to be predicate abstraction and use as our solver the $\texttt{VS}^3_{\texttt{PA}}$ tool.

The sortedness specification consists of the precondition $F_{\texttt{pre}} \doteq \texttt{true}$ and the postcondition $F_{\texttt{post}} \doteq \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1]$. The full functional specification would also ensure that the output array is a permutation of the input, but verifying—and thus, synthesizing—the full specification is outside the capabilities of most automated tools today.

We therefore use a mechanism to limit the space of programs to desirable sorting algorithms, while still only using $F_{\texttt{post}}$. We limit $D_{\texttt{exp}}$ to include only those operations that maintain elements—for example, *swapping* elements or *moving* elements to unoccupied locations. Using this mechanism, we ensure that invalid algorithms (that replicate or lose array elements) are not considered.

Non-recursive sorting algorithms Consider comparison-based sorting programs that are composed of nested loops. We specify a flowgraph template $R_{\texttt{flow}} \doteq *(*(\circ))$ and a computation template $R_{\texttt{comp}}$ that limits the operations to swapping of array values.

- $R_{\texttt{stack}} \doteq \emptyset$: The synthesizer produces two sorting programs that are valid with respect to the scaffold. One corresponds to Bubble Sort and the other is a non-standard version of Insertion Sort. The standard version of Insertion Sort

```
SelSort(int $A[], n$) {
 $i_1$:=0;
 while$^{\tau_1,\varphi_1}$($i_1 < n-1$)
    $v_1$:=$i_1$;
    $i_2$:=$i_1$+1;
    while$^{\tau_2,\varphi_2}$($i_2 < n$)
       if ($A[i_2] < A[v_1]$)
             $v_1$:=$i_2$;
       $i_2$++;
    swap($A[i_1], A[v_1]$);
    $i_1$++;
 return $A$;
}
```

Ranking functions:
$\varphi_1 : n - i_1 - 2$
$\varphi_2 : n - i_2 - 1$
Invariant τ_1:
$\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \land \ k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]$
Invariant τ_1:
$i_1 < i_2 \land i_1 \leq v_1 < n$
$\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \land \ k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]$
$\forall k : i_1 \leq k < i_2 \land k \geq 0 \Rightarrow A[v_1] \leq A[k]$

Figure 4.4: Synthesis result for Selection Sort. For ease of presentation, we omit degenerate conditional branches, i.e. true/false guards, We name the loop iteration counters $L = \{i_1, i_2, ..\}$ and the temporary stack variables $T = \{v_1, v_2, ..\}$.

uses a temporary variable to hold the inserted object. Since we do not provide a temporary variable, the synthesized program moves the inserted element by swapping it with its neighbor, while still performing operations similar to Insertion Sort.

- $R_{\texttt{stack}} \doteq \{(\texttt{int}, 1)\}$: The synthesizer produces another sorting program that uses the temporary variable to hold an array index. This program corresponds to Selection Sort and is shown in Figure 4.4. Notice the non-trivial invariants and ranking functions that are synthesized alongside for each of the loops.

Recursive divide-and-conquer sorting Consider comparison-based sorting programs that use recursion. We make a few simple modifications to the system to specify recursive programs. First, we introduce a terminal string $\circledast$ to the flowgraph template language, representing a recursive call.[2] Let $(F_{\texttt{pre}}(\vec{v_{\texttt{in}}}), F_{\texttt{post}}(\vec{v_{\texttt{out}}}))$ denote the functional specification. Then we augment the expansion to handle the new flowgraph string as follows:

$$\texttt{Expand}^n(\circledast) \quad = \quad \texttt{choose}\{[]\texttt{true} \rightarrow s_{\texttt{recur}}\}$$

where $s_{\texttt{recur}} = s_{\texttt{args}} \wedge (F_{\texttt{pre}}(\vec{v_{\texttt{in}}}') \Rightarrow F_{\texttt{post}}(\vec{v_{\texttt{out}}}'')) \wedge s_{\texttt{ret}}$ sets values to the arguments of the recursive call (using $s_{\texttt{args}}$), assumes the effect of the recursive call (using $F_{\texttt{pre}}(\vec{v_{\texttt{in}}}') \Rightarrow F_{\texttt{post}}(\vec{v_{\texttt{out}}}'')$, with the input arguments renamed to $\vec{v_{\texttt{in}}}'$ and the return variables renamed to $\vec{v_{\texttt{out}}}''$) and lastly, outputs the returned values into program variables (using $s_{\texttt{ret}}$). The statements $s_{\texttt{args}}, s_{\texttt{ret}}$ take the form:

$$\begin{aligned} s_{\texttt{args}} &= \textstyle\bigwedge_i x_i = e_i \quad \text{where } x_i \in \vec{v_{\texttt{in}}}', e_i \in D_{\texttt{exp}}|_{\texttt{Vars}} \\ s_{\texttt{ret}} &= \textstyle\bigwedge_i x_i = e_i \quad \text{where } x_i \in \texttt{Vars}, e_i \in D_{\texttt{exp}}|_{\vec{v_{\texttt{out}}}''} \end{aligned}$$

Here `Vars` denote the variables of the procedure (the input, output and local stack variables). We also tweak the statement concretization function to output a recursive call statement `rec`:

$$\texttt{Stmt}(F_{\texttt{pre}}(\vec{v_{\texttt{in}}}') \Rightarrow F_{\texttt{post}}(\vec{v_{\texttt{out}}}'')) = \vec{v_{\texttt{out}}}'' := \texttt{rec}(\vec{v_{\texttt{in}}}')$$

We specify a computation template that allows only swapping or moving of elements. We then try different values of the flowgraph and stack templates:

[2] Our notation has agreeable symmetry in that it denotes implicit iteration using acyclic fragments—hence the combination of $\circ$ and $*$.

- $R_{\texttt{flow}} \doteq \circledast;\circledast;\circ$ (two recursive calls followed by an acyclic fragment) and $R_{\texttt{stack}} \doteq \emptyset$: The synthesizer produces a program that recursively sorts subparts and then combines the results. This corresponds to Merge Sort.

- $R_{\texttt{flow}} \doteq \circ;\circledast;\circledast$ (an acyclic fragment followed by two recursive calls) and $R_{\texttt{stack}} \doteq \{(\texttt{int}, 1)\}$: The synthesizer produces a program that partitions the elements and then recursively sorts the subparts. This corresponds to Quick Sort.

4.5.4 Dynamic Programming Algorithms

For this category, we pick $D_{\texttt{prf}}$ to be predicate abstraction and use as our solver the $\texttt{VS}^3_{\texttt{AX}}$ tool. We choose all the textbook dynamic programming examples [71] and attempt to synthesize them from their functional specifications.

The first hurdle (even for verification) for these algorithms is that the meaning of the computation is not easily specified. To address this issue, we need support for axioms, which are typically recursive definitions.

Definitional Axioms Our tool $\texttt{VS}^3_{\texttt{AX}}$ allows the user to define the meaning of a computation as an uninterpreted symbol, with (recursive) quantified facts defining the semantics of the symbol axiomatically. For example, the semantics of Fibonacci are defined in terms of the symbol Fib and the axioms:

$$\texttt{Fib}(0) = 0$$

$$\texttt{Fib}(1) = 1$$

$$\forall k : k \geq 0 \Rightarrow \texttt{Fib}(k + 2) = \texttt{Fib}(k + 1) + \texttt{Fib}(k)$$

The tool passes the given symbol and its definitional axioms to the underlying theorem prover (Z3 [86]), which assumes the axioms before every theorem proving query. This allows the tool to verify dynamic programming programs.

Even with verification in place, automatic synthesis of these programs involves three non-trivial tasks for the synthesizer. First, the synthesizer needs to automatically discover a strategy for translating the recursion (in the functional specification) to *non-recursive iteration* (for the actual computation). The functional specifications do not contain this information, e.g., in the specification for Fibonacci above, the iteration strategy for the computation is not evident. Second, the synthesizer needs to take the (non-directional) equalities in the specifications and *impose directionality* such that elements are computed in the right order. For example, for Fibonacci the synthesizer needs to automatically discover that $\texttt{Fib}(k)$ and $\texttt{Fib}(k+1)$ should be computed before $\texttt{Fib}(k+2)$. Third, the synthesizer needs to discover an *efficient memoization* strategy for only those results needed for future computations, to fit the computation in the space provided—which is one of the benefits of dynamic programming algorithms. A naive hashmap-based strategy for memoization wastes space. On the other hand, if the synthesizer is able to infer the pieces of the computation required in the future, just from the recursive functional definition, then it can selectively overwrite old results and optimize the space required. For example, Fibonacci can be computed using only two additional memory locations by suitable memoization. Fortunately, just by specifying the resource constraints and using our proof-theoretic approach the synthesizer is able to perform these tasks and synthesize dynamic programming algorithms from their recursive functional specifications.

Also, as in the case of sorting, we want to disallow completely arbitrary computations. In sorting, we could uniformly restrict the expression language to only swap and move operations. For dynamic programming, the specification of the operations is problem-specific. For instance, for shortest path, we only want to allow the path matrix updates that correspond to valid paths, e.g., disallow arbitrary multiplication of path weights. $R_{\texttt{comp}}$ specifies these constraints by only permitting updates through certain predicates.

Dynamic programming solutions typically have an initialization phase (init-loop) and then a phase (work-loop) that fills the appropriate entries in the table. Therefore, we chose a $R_{\texttt{flow}}$ with an init-loop $(*(\circ))$ followed by a work-loop.

By specifying a flowgraph template $R_{\texttt{flow}} \doteq *(\circ);*(\circ)$ and a stack template with no additional variables (except for the case of Fibonacci, where the synthesizer required $R_{\texttt{stack}} \doteq \{(\texttt{int}, 2)\}$), we were able to synthesize the following four examples:

Fibonacci Consider computing the nth Fibonacci number from the functional specification as above. Our synthesizer generates a program that memoizes the solutions to the two subproblems $\texttt{Fib}(i_1)$ and $\texttt{Fib}(i_1 + 1)$ in the i_1th iteration. It maintains a sliding window for the two subproblems and stores their solutions in the two additional stack variables. The synthesized program along with its invariant and ranking function is shown in Figure 4.5.

```
Fib(int $n$) {
  $v_1$:=0;$v_1$:=1;$i_1$:=0;
  while$^{\tau,\varphi}$($i_1 \leq n$)
    | $v_1$:=$v_1$+$v_2$;swap($v_1, v_2$);
    | $i_1$++;
  return $v_1$;
}
```

Ranking function φ:
$x - s$
Invariant τ:
$v_1 = \texttt{Fib}(i_1) \wedge v_2 = \texttt{Fib}(i_1{+}1)$

Figure 4.5: Synthesis results for a dynamic programming program, Fibonnaci. Here, we name the loop iteration counters $L = \{i_1, i_2, ..\}$ and the temporary stack variables $T = \{v_1, v_2, ..\}$.

Checkerboard Consider computing the least-cost path in a rectangular grid (with costs at each grid location), from the bottom row to the top row. The functional specification states the path cost for a grid location in terms of the path costs for possible previous locations (i.e., below left, below, or below right). Our synthesizer generates a program that finds the minimum cost paths.

Longest Common Subsequence (LCS) Consider computing the longest common substring that appears in the same order in two given input strings (as arrays of characters). The recursive functional specification relates the cost of a substring against the cost of substrings with one fewer character. Our synthesizer generates a program for LCS.

Single Source Shortest Path Consider computing the least-cost path from a designated source to all other nodes where the weight of edges is given as a cost function for each source and destination pair. The recursive functional specification states the cost structure for all nodes in terms of the cost structure of all nodes if one fewer hop is allowed. Our synthesizer generates a program for the single source shortest path problem.

For the following two examples, synthesis failed with the simpler work-loop, but we synthesize the examples by specifying a flowgraph template $*(\circ);*(*(\circ))$ and no additional stack variables:

All-pairs Shortest Path Consider computing all-pairs shortest paths using a recursive functional specification similar to the one we used for single source shortest path. Our synthesizer times out for this example. We therefore attempt synthesis by (i) specifying the acyclic fragments and synthesizing the guards, and (ii) specifying the guards and synthesizing the acyclic fragments. In each case, our synthesizer generates the other component, corresponding to Floyd-Warshall's algorithm.

Matrix Chain Multiply Consider computing the optimal way to multiply a matrix chain. Depending on the bracketing, the total number of multiplications varies. We wish to find the bracketing that minimizes the number of multiplications. E.g., if we use the simple n^3 multiplication for two matrices, then $A_{10\times100}B_{100\times1}C_{1\times50}$ can either takes 1,500 multiplications for $(AB)C$ or 55,000 multiplications for $A(BC)$. The functional specification defines the cost of multiplying a particular chain of matrices in terms of the cost of a chain with one fewer element. Our synthesizer generates a program that computes the optimal matrix bracketing.

4.5.5 Performance

Table 4.1 presents the performance of a satisfiability-based synthesizer over arithmetic, sorting and dynamic programming benchmarks. All runtimes are median of three runs, measured in seconds. We measure the time for verification and the time for synthesis using the same tool. The total synthesis time varies between 0.12 and 9658.52 seconds, depending on the difficulty of the benchmark, with a median runtime of 14.23 seconds. The factor slowdown for synthesis varies between 1.09 and 92.28, with a median of 6.68.

The benchmarks we used are considered difficult even for verification. Consequently the low average runtimes for proof-theoretic synthesis are encouraging. Also, the slowdown for synthesis compared to verification is acceptable, and shows that we can indeed exploit the advances in verification to our advantage for synthesis.

4.5.6 Discussion

The synthesis of the expressive programs reported in this chapter is made feasible by the use of some simplifying ideas that we discuss here.

Array flattening Two (and higher) dimensional arrays, while making it easier for human programmers to reason about data, and indices into it, have little semantic benefit over one dimensional arrays. E.g., instead of indexing an 2D-array using

[3] These timings are for separately (i) synthesizing the loop guards, and (ii) synthesizing the acyclic fragments. We fail to synthesize the entire program, but with these hints provided by the user, our synthesizer can produce the remaining program.

	Benchmark	Verif.	Synthesis	Ratio
Arith. ($\texttt{VS}^3_{\texttt{QA}}$)	Swap two	0.11	0.12	1.09
	Strassen's	0.11	4.98	45.27
	Sqrt (linear search)	0.84	9.96	11.86
	Sqrt (binary search)	0.63	1.83	2.90
	Bresenham's	166.54	9658.52	58.00
Sorting ($\texttt{VS}^3_{\texttt{PA}}$)	Bubble Sort	1.27	3.19	2.51
	Insertion Sort	2.49	5.41	2.17
	Selection Sort	23.77	164.57	6.92
	Merge Sort	18.86	50.00	2.65
	Quick Sort	1.74	160.57	92.28
Dynamic Prog. ($\texttt{VS}^3_{\texttt{AX}}$)	Fibonacci	0.37	5.90	15.95
	Checkerboard	0.39	0.96	2.46
	Longest Common Subseq.	0.53	14.23	26.85
	Matrix Chain Multiply	6.85	88.35	12.90
	Single-Src Shortest Path	46.58	124.01	2.66
	All-pairs Shortest Path[3]	112.28	(i) 226.71 (ii) 750.11	(i) 2.02 (ii) 6.68

Table 4.1: Experimental results for proof-theoretic synthesis over different domain. (a) Arithmetic (b) Sorting (c) Dynamic Programming. For each category, we indicate the tool used to solve the verification conditions and the synthesis conditions.

a pair (i, j), a semantically identical 1D-array can be used, indexed by an integer $i * rowsize + j$. To the synthesizer, or in pseudocode, these representations are essentially identical, and we can arbitrarily pick the one more convenient. The theory of arrays is more conveniently defined over flatted arrays and therefore our synthesized programs are in that representation. This also removes some arbitrary non-determinism (in the space of programs) and simplifies control flow (instead of nested iteration counters, a single iteration counter suffices). Array flattening also facilitates abstracting layout non-determinism for dynamic programming examples as described below.

Abstracting layout Most benchmarks for dynamic programming memoize results to subproblems by filling a table. Aside from some causal constraints there is little definedness in the order in which entries are filled out. Thus there is no one unique way of laying out the entries in the table.

For programs that manipulate a two (or higher) dimensional table, we realize that the layout of the entries is immaterial as long as some ordering constraints are maintained amongst the entries. For example, a program that traverses the top-left half-triangle of a square matrix using diagonals can be rewritten as a program that traverses the bottom-left half-triangle using row-wise traversals. Both of these can in turn be rewritten as straight-line traversal over a one-dimensional array as well, i.e. the layout can be flattened.

We let the user specify the layout constraints again over *uninterpreted layout functions* and synthesize the program over these abstract layout functions. Note that now, the definitional axioms also have to be defined using the layout functions. The constraints over the layout functions can later be used to synthesize arbitrary concrete layouts to get executable programs. For example, the layout constraint for a program that does a diagonal traversal of the top-left triangle of a square matrix is defined in terms of functions $\texttt{up}, \texttt{left}$ and the constraints $\texttt{up}(x) < x \wedge \texttt{left}(x) < x \wedge \texttt{up}(\texttt{left}(x)) < x \wedge \texttt{left}(\texttt{up}(x)) < x$. Once a program has been synthesized in terms of these functions, a simple theorem proving query can find a satisfying concretization to the functions and the program can be rewritten as a two dimensional traversal. This theorem proving query is a simple $\exists$ query to find a satisfying solution for the layout constraints. One way to formulate the $\exists$ query would be to use templates for the abstract functions and solve for coefficients similar to our approach in Chapter 2. For example, if n is the dimension of the matrix then $\texttt{left}(x) \doteq x - 1, \texttt{up}(x) \doteq x - n$ would be the natural concretization, but any other concretization satisfying the constraints would be valid too. For instance, one that traverses the bottom-left triangle in row-order.

Computational templates We now discuss how computational templates help restrict program operations to a desired space, alleviating the massive undertaking of verifying termination, and full functional correctness in a single step.

To synthesize programs that meet a given functional behavior, this chapter argues that, at least, one must be capable of verifying that behavior. Not only that, to be useful for synthesis the full functional verification needs to be done in one step, and cannot be done piecewise. While piecewise verification can verify partial programs properties by considering them in turn, piecewise synthesis may yield solutions that are mutually inconsistent and therefore irreconcilable. That said, there might be a way out.

While functional specifications define the computation theoretically, the results in this chapter indicate that other information, e.g., domain and resource restriction, can make the synthesis task practical. The use of resource constraints, specifically, the computational restrictions, removes the need to assert part of functional specifications in certain cases. For instance, for the case of sorting, while the full functional specification asserts that the output array is a permutation of the input, using the computational template we restrict array writes to `swap`s, eliminating the need to assert a permutation constraint. Similarly, for the case of dynamic programming benchmarks, we ensure that updating to the memoization table are through limited operations that do not violate core soundness, eliminating a need to assert part of the functional specification.

Choosing good *programs* The system described in this chapter does not attempt to attach a preferability metric to programs—except of course it *prefers* correct, well-formed, and terminating programs.

Theoretically, functional specifications capture the desired computation. But in practice, programmers also care about the resources (space and time) used by their programs and of the average case performance. There might be other concerns, such as particular memory access patterns, or ordered computations etc., that certain programmers, for instance, security aware developers, may care about.

There are three possibilities for synthesizing such *good* programs. First, we may leave it up to the programmer and have the automated tool just enumerate all valid solutions. This is the approach we currently follow. Second, we may encode domain specific constraints, e.g., limiting network communication, in addition to safety, termination, and well-formedness, to ensure the synthesizer only generates good programs. This is, in spirit, similar to our core technique presented in this chapter, and we will potentially pursue this next. Lastly, we may use an iterative mechanism on top of the core synthesizer to filter the good candidates which the tool enumerates. This is, in spirit, similar to previous work on Sketching [245] that enumerates candidate programs and uses a model checker to eliminate bad programs, where in their case, the bad programs are those that do not meet the safety criteria. This approach may indeed be the only plausible one for complicated performance issues such as optimizing cache performance that are hard to model as constraints.

Modular synthesis Our use of `swap` operations in sorting, layout functions in dynamic programming, and in general synthesizing programs over a given set of predicates, is an instance of synthesis with respect to an abstraction. Albeit, an abstraction that is user provided. Other authors have also explored the use of such provided abstractions to design *compositional* synthesis systems [152, 153].

In the future, we wish to design a synthesis system that automatically infers a suitable abstraction boundary and synthesizes functions in terms of the interface thus defined. [257]

4.6 Summary

This chapter presented a principled approach to synthesis that treats synthesis as a generalized verification problem. The novelty of this approach lies in generating synthesis conditions, which are composed of safety conditions, well-formedness conditions, and progress conditions, such that a satisfying solution to the synthesis conditions corresponds to a synthesized program. We used verification tools $\mathtt{VS}^3_{\mathtt{LIA}}$ and $\mathtt{VS}^3_{\mathtt{PA}}$ from previous chapters to synthesize programs, and, simultaneously, their proof (invariants, ranking functions). We demonstrated the viability of our approach by synthesizing difficult examples in the three domains of arithmetic, sorting, and dynamic programming, all in very reasonable time.

4.7 Further Reading

Deductive Synthesis Deductive synthesis is an approach to synthesis that generates programs through iterative refinement of the specification. At each step of the refinement, well-defined proof rules are used, each of which corresponds to the introduction of a programming construct. For instance, case splits in the proof leads to a conditionals in the program, induction in the program leads to loops in the program. Deductive synthesis was explored in work by Manna, Waldinger and others in the 1960's and 1970's [195]. The core idea was to *extract* the program from the *proof* of realizability of the formula $\forall \vec{x} : \exists \vec{y} : pre(\vec{x}) \Rightarrow post(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ are the input and output vectors, respectively [129, 263].

The approach presented here can be seen as automating deductive synthesis.

Additionally, the technical insight added by this dissertation is the realization that while human-guided proof-refinement implicitly steers away from pathological cases, when automating the process, a critical requirement is to ensure well-formedness and termination, in addition to refining the safety proof.

Alternative exciting directions The interested reader is also advised to follow the developments by other independent groups of techniques that are similar in spirit, i.e., are automatic and deductive, but differ in technical content. Of particular interest is the work by Vechev, Yahav and Yorsh [258] on iterative refinement of the proof *and* program. Another exciting direction is the work by Kuncak's group on decision procedures for program synthesis [171, 197] to be incorporated into custom solvers.

Chapter 5

Path-based Inductive Synthesis: Testing-inspired Program Synthesis

"I don't know what my path is yet.
I'm just walking on it."
— Olivia Newton-John[1]

This chapter describes a novel technique that synthesizes programs using an approach inspired by, and providing approximate guarantees as in, testing. Our approach combines ideas from symbolic execution-based testing and satisfiability-based program analysis.

We describe the technique as working over a template of the program to be synthesized. For the applications we consider we find that we can automatically *mine* this template. The mined template finitizes the space of programs, but the space is still exponential. To efficiently find a solution, we propose a technique that iteratively prunes away invalid programs from the search space. We pick a candidate solution to the synthesis and identify a feasible path for the candidate through the template. Using ideas from satisfiability-based analysis we find potential solutions

[1] English-born, Australian raised singer/actress and an environmental, animal rights, and breast cancer activist.

that satisfy the specification for the set of paths accumulated. We continue this Path-based Inductive Synthesis (PINS) procedure until the space contains only valid inverses.

We apply PINS to the problem of automatically generating *program inverses*, i.e., of synthesizing a program P^{-1} that negates the computation of a given program P. This problem arises naturally in paired computations such as compression-decompression, serialization-deserialization, transactional memory rollback, and bidirectional programming. Automatic program inversion can alleviate the cost associated with maintaining two closely related programs, and ensure correctness and maintainability. We make two observations. First, we observe that the control flow structure of the inverse is very similar to the original program. Therefore we can automatically mining the flowgraph, expression and predicate sets from the original program. Our approach limits user effort to simple modifications of the mined template, if at all required. Second, we observe that the specification of inversion is trivial (identity) when we consider the concatenation (sequential composition) of the original program with its inverse. We apply PINS to the template formed by the concatenation of the original known program with the mined unknown program to synthesize inverses.

We also apply PINS to the problem of automatically generating *paired network programs*, such as clients from servers or vice versa. We exploit the observation that for these paired programs, the desired program has a control flow structure very related to its pair, and the expressions and guards are related as well. From the original program, we syntactically mine the control flow structure, expression and guard sets for its pair—that we intend to synthesize—finitizing the problem and then apply PINS to synthesize valid solutions.

Using PINS, we show we can synthesize inverses for compressors (e.g., LZ77), packers (e.g., UUEncode), and arithmetic transformers (e.g., image rotations). Additionally, we use PINS to synthesize a TFTP client from its server. These programs (and their corresponding pairs) range from 5 to 20 lines of code, and PINS synthesizes them in a median time of 40 seconds, demonstrating the viability of our synthesis approach.

5.1 Using Symbolic Testing to Synthesize Programs

Testing can be considered as an approximation to formal verification. In a similar vein, we intend to develop a synthesis technique with approximate guarantees. While the approach in the previous chapter provides formal guarantees, it does so by inferring program invariants, which may be complicated. The approach in this chapter does not provide formal guarantees, but can synthesize programs without reference to invariants. The added expense of inferring proofs may be justified when synthesizing critical software, but in this chapter we consider the case where the proof is only of auxiliary importance, and we wish the technique to automatically generate complicated, hard to maintain, programs.

The approach in this chapter does not rely on formal verifiers, unlike proof-theoretic synthesis and as some other previous approaches do [245]. Formal verifiers are hard to build, are domain specific, and for the programs we target in this chapter, we do not know of any tools that can formally verify them correct. On the other hand, testing, and in our case symbolic testing [165] (Section 5.3), has been shown to be a good (approximate) verification strategy—perhaps the only one in the absence of formal verifiers—and therefore can potentially be employed for synthesis. Our technique, called `PINS`, consists of the following steps:

Step 1 *(Finitize the problem)* We finitize the problem by constructing a flowgraph template with placeholders for guards and expressions, and a set of potential expression and predicate sets for those placeholders. While `PINS` is a general synthesis technique that works over a given template flowgraph and expressions and predicate sets, for the case of our application, i.e., inversion, we will be able to mine the program `Prog` to get the flowgraph and expression and predicate sets for P^{-1}.

Step 2 *(Encode correctness constraints using paths)* We use symbolic execution to generate correctness (safety and termination) constraints over a set of paths through the template program. We then use SMT reasoning to convert these constraints into concise SAT constraints which we solve for candidate solutions. These candidate solutions satisfy all the correctness constraints for those paths.

Step 3 *(Refine solution space)* We generate new feasible paths for some candidate solution that we generated in Step 2. Note that a candidate program may not be a valid program for the synthesis task as it is only correct up to the set of paths explored until that point. Therefore, we generate a new path parameterized by this candidate solution. Our novel path construction works without a formal verifier, and instead of attempting to find a counterexample it generates paths that reinforce valid solutions and are likely to eliminate invalid solutions.

Step 4 *(Repeat 2,3 until stabilized)* We iteratively use Steps 2 and 3 to refine the space of candidate solutions until only valid ones remain.

The distinguishing feature of our approach is that at no point do we try to formally *prove* that a candidate solution is actually a valid synthesis solution. Instead, our approach is more like symbolic testing: we try to find a set of paths that provide sufficient witness that our candidates are indeed valid. More precisely, let *sols* be the set of solutions we output after stabilization. Then for each $S \in sols$, the corresponding candidate program is indeed valid on every path explored, i.e., it met the specification on each of the explored paths. Analogously, for every $S \notin sols$ that the algorithm discarded during iteration, at least one path was explored that shows that S violates the specification. Once we have sufficient coverage, there is only a small chance that the resulting solution is not a true solution to the synthesis problem—and in our experience, `PINS` is able to refine the search space down to a single valid program most of the time (Section 5.4).

We apply `PINS` to the problem of automatic program inversion [89, 130, 54, 101, 119] (Section 5.5). Specifically, we consider inverting an injective program `Prog` by finding another program P^{-1} that is its left inverse.

```
Prog
in($A, n$)
assume($n \geq 0$);
$i$:=0; $j$:=0;
while ($i < n$)
   $r$:=0;
   while ($A[i] = 0$)
      $r$++; $i$++;
   if ($r = 0$)
      $B[j$++]:=$A[i$++];
   else
      $B[j$++]:=$-r$;
out($B, j$)
```

(a)

Input Stream	*Output Stream*
0	
0	
0	−3
1	1
1	1
2	2
0	−1
3	4
0	−2
0	1
1	

(b)

Figure 5.1: Illustrating PINS using an example. (a) A program that compresses runs of a special integer "0" in an array with non-negative integers (b) An input array with its corresponding compressed output.

5.2 Motivating Example and Technical Overview

In this section, we illustrate PINS using an example. Consider the program Prog shown in Figure 5.1(a). Prog compresses a particular frequently occurring integer designated by 0. This is in fact a simplified version of a full run-length encoder, which our technique can also invert (Section 5.6). In the outer loop the program processes the integers of the input array A, of length n, and in each iteration Prog counts the number of occurrences r of the special integer. If the count is non-zero then it outputs the count (negated to distinguish it from the other positive integers) to the output array B. If the count is zero it copies the non-zero integer as-is to the output array B. Figure 5.1(b) shows an example input array and corresponding output array.

Now suppose, for the moment, that the user specifies a *flowgraph template* *fg*, shown in Figure 5.2(a), for the inverse. A flowgraph template consists of con-

$\boxed{fg}$

```
in($B, j$)
$\langle i', j' := \varepsilon_1, \varepsilon_2 \rangle$
while ($\rho_1$)
    $\langle \underline{r' := \varepsilon_3} \rangle$
    if ($\rho_2$)
        $\langle A', i', j' := \varepsilon_4, \varepsilon_5, \varepsilon_6 \rangle$
    else
        $\langle r', j' := \varepsilon_7, \varepsilon_8 \rangle$
    while ($\rho_3$)
        $\langle A', r', i' := \varepsilon_9, \varepsilon_{10}, \varepsilon_{11} \rangle$
out($A', i'$)
```

(a)

$\boxed{\Pi_e}$

$$\left\{ \begin{array}{c} 0, r'-1, -\text{sel}(B, j'), \\ \text{upd}(A', i', \text{sel}(B, j')), \\ \underline{i'+1}, \underline{j'+1}, \underline{-1} \end{array} \right\}$$

$\boxed{\Pi_p}$

$$\left\{ \begin{array}{c} \underline{B[j'] > 0}, j' < j, \\ r' > 0, i' > 0 \end{array} \right\}$$

(b)

$\boxed{P^{-1}}$

```
in($B, j$)
$i'$:=0; $j'$:=0;
while ($j' < j$)
    $r'$:=0;
    if ($B[j'] > 0$)
        $A'[i'$++]:=$B[j'$++];
    else
        $r'$:=$-B[j'$++];
    while ($r' > 0$)
        $A'[i'$++]:=0; $r'$--;
out($A', i'$)
```

(c)

Figure 5.2: Ilustrating PINS using an example: (a) The flowgraph template *fg* for synthesis (b) The expression set Π_e and predicate set Π_p for synthesis (c) The synthesized inverse, which is *fg* instantiated with a solution $S = \{\varepsilon_1 \mapsto 0, \varepsilon_2 \mapsto 0, \rho_1 \mapsto \{j' < j\}, \varepsilon_3 \mapsto 0 \,..\}$.

trol flow structures, guarded with unknown predicates ρ_i's, and parallel assignment blocks $\langle x_1, x_2 \,.. := \varepsilon_1, \varepsilon_2, ..\rangle$, indicating that the variables $x_1, x_2, ..$ are assigned the unknown expression $\varepsilon_1, \varepsilon_2, ..$, respectively. Parallel assignment ensures that we can ignore the order in which the variables are assigned in a basic block (as described in Chapter 4). Also suppose, for the moment, that the user specifies a candidate *predicate set* Π_p and *expression set* Π_e (Figure 5.2(b)) that can be used to instantiate the ρ's and ε's, respectively. Such user-provided sets are standard, as in the previous chapter and in other approaches to synthesis [245] and predicate abstraction-based verification [128]. We shall see later that for program inversion the sets can almost entirely be mined from the original program, alleviating the human effort involved in guessing these sets.

Notice that the above only finitizes the solution space, but efficiency solving for an inverse is still not easy. Even for this small flowgraph, with 10 holes that range over 7 expressions and 3 holes that range over subsets (conjunctions) of 4 predicates, the space of possible inverses has $7^{10} \times (3 \times 2^4) \approx 2^{34}$ solutions if types are ignored and $6^9 \times (3 \times 2^4) \approx 2^{29}$ otherwise. Therefore a naive exhaustive search for a solution will not work, and so we describe a strategy that implicitly categories solutions and constructs symbolic paths that prune invalid categories of solutions iteratively.

Solving for the inverse using directed symbolic testing Given the flowgraph *fg*, predicate set Π_p, and expression set Π_e, we now describe a path-based approach that can synthesize the inverse P^{-1}.

We use *symbolic execution* to generate safety and termination constraints through the composition $\texttt{Prog} \circ fg$ of the original and the template flowgraph. Symbolic testing allows us to generate the constraints without needing complicated loop invariants. We reduce these constraints to *SAT constraints* using techniques that we described in Chapter 3. We then solve the SAT instance to get candidate inverses. For instance, one path, through $\texttt{Prog} \circ fg$ is $n \geq 0; i := 0; j := 0; i \geq n; \langle i', j' := \varepsilon_1, \varepsilon_2 \rangle; \neg \rho_1$. This path generates the safety constraint:

$$\exists E \forall X : \left(\begin{array}{c} n^0 \geq 0 \wedge i^1 = 0 \wedge j^1 = 0 \wedge i^1 \geq n^0 \ \wedge \\ i'^2 = {\varepsilon_1}^{V_1} \wedge j'^2 = {\varepsilon_2}^{V_1} \ \wedge \neg {\rho_1}^{V_2} \end{array} \right) \Rightarrow id$$

$$\text{with } id \ \doteq \ (n^0 = i'^2) \wedge (\forall k : 0 \leq k < n^0 \Rightarrow A^0[k] = A'^0[k])$$

where E and X are the set of unknowns $\{\varepsilon_1, \varepsilon_2, \rho_1\}$ and the set of program variables $\{n^0, i^1, j^1, i'^2, j'^2\}$, respectively. The integer superscripts denote the version numbers of the program variables. Each assignment to a program variable increments its version number. Unknowns are superscripted with *version maps* from program variables to their latest version. When an unknown is instantiated, the variables in the substitution are lifted to the versions specified by the map. For example, for ${\varepsilon_1}^{V_1}$ the version map is $V_1 = \{n \mapsto 0, i \mapsto 1, \ldots\}$. So if ${\varepsilon_1}^{V_1}$ is instantiated with the expression $i - n + 1$, the result is $i^1 - n^0 + 1$ Also, the identity fact *id* is syntactically generated from the annotations $\texttt{in}(A, n)$ and $\texttt{out}(A', i')$ with the fact that A and A' are arrays with lengths n and i', respectively.

Notice that from the first line in the antecedent we get $n^0 \geq 0 \wedge (i^1 = 0 \geq n^0)$, which implies $n^0 = 0$. Therefore the quantified fact in *id* is trivially satisfied, but to prove $n^0 = i'^2$ we need $\varepsilon_1^{V_1}$ to be 0. The only expression from Π_e that we can assign to ε_1 to ensure this is 0 (and then $\varepsilon_1^{V_1}$ will be 0 too).

`PINS` solves such constraints using the technique described in Section 3.6.2 that converts the above SMT constraints into SAT constraints over boolean indicator variables $b_{\varepsilon \mapsto \bar{\varepsilon}}$ (i.e., Eq. 3.7). That indicator variable being assigned to true in a solution corresponds to unknown ε having the expression $\bar{\varepsilon} \in \Pi_e$. Thus, the SAT instance generated from the current path will contain the clause with the sole literal:

$$(b_{\varepsilon_1 \mapsto 0}) \tag{5.1}$$

i.e., saying "unknown ε_1 must map to 0."

But notice that given our expression and predicate sets, this is not the only way to satisfy the safety constraint above. We could also make the antecedent `false`, which happens under the assignment $\varepsilon_2 \mapsto -1$ and $\rho_1 \mapsto \{j' < j\}$. (Note that expressions map to single values while predicates map to subsets indicating conjunction.) Under these assignments the antecedent contains $j^1 = 0 \wedge j'^2 = -1 \wedge \neg(j'^2 < j^1)$, which simplifies to `false` and therefore satisfies the constraint trivially. Thus, the SAT instance `PINS` solves will actually have the above clause disjuncted with the following (and others cases that result in `false`):

$$(b_{\varepsilon_2 \mapsto -1} \wedge b_{\rho_1 \mapsto j' < j}) \tag{5.2}$$

Clause (5.2) does not constrain ε_1 at all. If the solution map from these is used to instantiate *fg*, we see that clause (5.2) allows solutions that correspond to invalid inverses. On the other hand, $\varepsilon_0 \mapsto 0$ is part of the solution for a true *valid* inverse. Therefore the next step is to add paths to constrain the indicators further to eliminate Eq. 5.2 while leaving Eq. 5.1 as the only possibility.

We could use random path exploration to find new paths, but in practice we have found that approach fails to converge in a reasonable amount of time. `PINS` therefore uses a novel path construction algorithm that, given a solution S, finds a path that is expected to be relevant to S.

Let $[\![fg]\!]S$ stand for the instantiation of *fg* with S. Given S, we use symbolic execution to find a new, *feasible* path through `Prog`$\circ([\![fg]\!]S)$, meaning one such that the antecedent of the safety constraint is not false. By forcing the path to be feasible, we constrain the search space so that any remaining solutions have a reasonable number of feasible paths over which they satisfy the specification. In contrast, random path exploration tends to generate paths that are infeasible. (Notice that we are solving for the inverse program as part of this process, so we cannot a priori identify the feasibility of a path without fixing a particular S.)

For example, consider an invalid solution $S_{Eq.5.2} = \{\varepsilon_2 \mapsto -1, \rho_1 \mapsto \{j' < j\}\} \cup S'$ that is allowed by Eq. 5.2, where S' assigns any value to the remaining unknowns, lets say, $S' \doteq \{\varepsilon_1 \mapsto 0, \varepsilon_3 \mapsto 0, \varepsilon_4 \mapsto \texttt{upd}(A', i', \texttt{sel}(B, j')),\ \varepsilon_5 \mapsto i' + 1, \varepsilon_6 \mapsto j' + 1, \rho_2 \mapsto \{B[j'] > 0\}, \rho_3 \mapsto \{r' > 0\}, ..\}$. Since $j \geq 0$ at the end of the original program, all feasible paths will enter the outer loop of the inverse at least once for this solution. Specifically, one path is $n \geq 0; i := 0; j := 0; i \geq n; \langle i', j' := \varepsilon_1, \varepsilon_2\rangle; \rho_1; r' := \varepsilon_3; \rho_2; \langle A', i', j' := \varepsilon_4, \varepsilon_5, \varepsilon_6\rangle; \neg\rho_3; \neg\rho_1$. If we substitute $S_{Eq.5.2}$ into the constraint generated we find that $i' = 1$ and $n = 0$ at the end of the path, and so the safety assertion requiring their equality is violated. Additionally, the antecedent of the constraint does not imply `false` by construction. Therefore, adding the constraint corresponding to this path eliminates $S_{Eq.5.2}$. Notice that this path is only feasible *with respect to this solution*, and in particular, infeasible for any valid inverse. So in synthesis even infeasible paths (with respect to valid inverses) help prune the search space, as long as they are chosen carefully.

Iteratively refining the space using directed path generation as above yields a constraint satisfied by solutions with a reasonable number, typically less than 15-20, of feasible paths for each. In our example, this iterative process yields the valid inverse P^{-1}, shown in Figure 5.2(c).

Mining the template of the inverse For the kind of non-trivial inverses we intend to synthesize, we find that the flowgraph, expression, and predicate sets are difficult for the user to guess from scratch. On the other hand, the often-mentioned approach of enumerating all possible predicates and expressions between program variables does not scale due to the large solution space in synthesis. Previous approaches (even our approach in the previous chapter and others [245]) do not provide any concrete suggestions about where to get the predicates from. Fortunately, we can exploit the structure of the inversion problem to *mine* these from the given program `Prog`. Our approach is inspired by Dijkstra's observation that at times, inverses are just the original program read backwards (the edges are reversed, variables read in `Prog` are assigned to in P^{-1}, and expressions in `Prog` are replaced by their "inverses" in P^{-1}). We find that not all inverses work this way. Occasionally, the flow of control in P^{-1} is in the same direction as in `Prog` (edges are not reversed, variables assigned are the same, and expressions have the same form), and at times blocks of statements need to be omitted. Instead of guessing the entire flowgraph, expression and predicate sets, we ask the user to just guess these forwards ↓, backwards ↑, or deletion × tags on the main control flow structures (loops, conditionals, and main entry point)—typically starting with all ↑ tags. For example, with tags of ↓, ↑, ↑, on the entry and two nested loops in Figure 5.1(a), we can mine values for fg, Π_e, and Π_p. If synthesis fails with the initial values the user makes minor tweaks (guided by the paths `PINS` explored for eliminating all solutions). Our mining heuristic yielded Figure 5.2(a,b)—with the minor user tweaks underlined. Notice that because of the ↑ tag on the outer loop, the order of the enclosed conditional and loop are correctly reversed.

5.3 Preliminaries

We now define the language of programs and our formalism for symbolic execution.

Language of Statements Our algorithm operates over programs with statements given by the following language:

$$\begin{array}{rcl} \mathit{stmt} & ::= & x := e \;\mid\; \texttt{assume}(p) \;\mid\; \mathit{stmt}; \mathit{stmt} \\ e & ::= & \bar{\varepsilon} \;\mid\; \varepsilon \\ p & ::= & \bar{\rho} \;\mid\; \rho \end{array}$$

The language consists of assignments $x := e$ between a variable x and an expression e, assume statements $\texttt{assume}(p)$ that take a predicate p, and the sequencing operator ';'. Expressions are either known $\bar{\varepsilon}$ or unknown symbols ε. Similarly, predicates are either known $\bar{\rho}$ or unknown symbols ρ. Known expressions come from a standard language $\bar{\varepsilon} ::= x \mid \mathit{ufs}(x) \mid \bar{\varepsilon}\ \mathit{op}\ \bar{\varepsilon} \mid \texttt{sel}(\bar{\varepsilon}, \bar{\varepsilon}) \mid \texttt{upd}(\bar{\varepsilon}, \bar{\varepsilon}, \bar{\varepsilon})$ with variables x, arithmetic operations *op*, array operators `sel` and `upd`, and uninterpreted function symbols *ufs*. Known predicates are pairs of known expressions separated by relational operators. For notational convenience, we may use the `skip` statement as well, which can be modeled in the language as `assume(true)`.

We will use `Prog` and $\widehat{\texttt{Prog}}$ to denote the known program and the unknown template program, respectively, and use *fg* to denote the unknown flowgraph for $\widehat{\texttt{Prog}}$. In our formalism this means that `Prog` and $\widehat{\texttt{Prog}}$ are in terms of known $\bar{\varepsilon}$ and $\bar{\rho}$ and *fg* is in terms of unknown ε and ρ. This formalism also allows us to freely mix statements of either form, in ways used by previous approaches to general synthesis [246]. Thus, even though we only apply PINS to inversion and client-server synthesis for demonstration, the idea of symbolic testing-based synthesis is generally applicable to automatically completing any partial program.

Programs and their composition Programs in our system consist of statements as above and control flow edges. We assume that the program is structured and does not contain arbitrary jumps, i.e., loops are well-formed and can be easily identified from the control flow graph. Our language contains assume statements and so, without loss of generality, we treat all branches as non-deterministic.

Aside from the language of statements and control flow structure, we need three other components to define the synthesis task:

$$\vec{v_{\texttt{in}}}, \vec{v_{\texttt{out}}} : \text{Vector of input and output variables, respectively}$$

$$\circ : \text{The compose operator: sequential ` ; ' or parallel ` || '}$$

$$\texttt{spec} : \text{Specification of the composed program, typically identity}$$

The operator $\circ$ will be used to compose the known and the unknown programs. Sequential composition ($\texttt{Prog}\ ;\ \widehat{\texttt{Prog}}$) indicates that Prog executes first with input values for $\vec{v_{\texttt{in}}}$, producing values for $\vec{v_{\texttt{out}}}$, followed by the execution of $\widehat{\texttt{Prog}}$, which takes values for $\vec{v_{\texttt{out}}}$ as input and in turn produces values for $\vec{v_{\texttt{in}}}$. Parallel composition ($\texttt{Prog} \mid\mid \widehat{\texttt{Prog}}$) indicates that Prog and $\widehat{\texttt{Prog}}$ together take input values for $\vec{v_{\texttt{in}}}$ and execute simultaneously, interacting using messaging primitives to produce values for $\vec{v_{\texttt{out}}}$.

Definition 5.1 (Synthesis task) *Given an known (terminating) program* Prog*, the* synthesis task *is to find another (terminating) program* $\widehat{\texttt{Prog}}$ *such that the fol-*

lowing Hoare triple is valid:

$$\{\texttt{true}\}\ \texttt{Prog} \circ \widehat{\texttt{Prog}}\ \{\texttt{spec}\} \tag{5.3}$$

Notice that for inversion—with sequential composition and with `spec` equal to identity—this definition is analogous to the left inverses of mathematical functions, i.e., $P^{-1}(\texttt{Prog}(x)) = x$. For parallel composition, the Hoare triple means that when simultaneously started with precondition `true`, both programs terminate in a combined state that satisfies `spec`.

Versioned variables and expressions We associate an integer *version* with each variable. The versions denote the different values taken by the variables at different points in time. A *versioned variable* x^v denotes the variable x at version v. This notion is extended to versioned predicates and expressions. A *versioned expression* e^V is the expression e with each variable x in it replaced with the versioned variable $x^{V[x]}$ at the version as given by the map V. This is straightforward for known expressions $\bar{\varepsilon}$, and for unknown expressions ε we delay assigning versions to variables until the unknown has been replaced with a known. Similarly, we define a *versioned predicate* p^V for a predicate p and version map V. We will use $V_{\texttt{init}}$ to denote an initial version map with $V_{\texttt{init}}[x] = 0$ for all variables x in the program.

Paths, Path Constraints and (In)feasibility A *path* in the program is a sequence of assignments or assume statements seen while following the control flow edges from the beginning to the end of the program. A *path constraint* or *trace* τ corresponding to a path is a conjunction of predicates that are either equality predicates for assignment statements, or boolean predicates for assume statements. Paths contain unversioned variables and expressions while path constraints contain versioned variables and expressions. Assume statements `assume`(p) on a path give rise to versioned predicates p^V in the corresponding path constraint. Assignment statements $x := e$ on the path give rise to an equality $x^v = e^V$ between the next version v of the variable x and the versioned expression e^V in the corresponding path constraint. We call a path *feasible* if its path constraint does not imply `false` and *infeasible* otherwise.

Solution Maps A *solution map* S is an assignment of unknown expressions ε and predicates ρ to known expressions $\bar{\varepsilon}$ and subset of predicates $\{\bar{\rho}_i\}_i$, respectively. (A subset of predicates $\{\bar{\rho}_i\}_i$ stands for their conjunction $\wedge_i \bar{\rho}_i$.) We define the notion of an *interpretation* $[\![fg]\!]S$ of an unknown program fg with respect to a solution map S as the program with its unknown expressions and predicates instantiated according to the map. We define a similar notion for unknown expressions $[\![\varepsilon]\!]S$, predicates $[\![\rho]\!]S$ (versioned or otherwise), and path constraints $[\![\tau]\!]S$. A solution map need not assign to all unknowns, in which case the unassigned unknowns remain unchanged.

Symbolic execution Given a program path we generate its path constraint using the operational semantics of a symbolic executor `SymEx` shown in Figure 5.3. For each statement in our language, the symbolic executor takes the *path constraint* τ and *version map* V up to that point and returns the updated path constraint and version map. The symbolic executor is parameterized by a solution map S and by a set of path constraints $\{\tau_i\}$. We will later use S to ensure that the path is feasible for that solution and use $\{\tau_i\}$ to indicate the set of paths that are to be avoided. For now, we can assume that the solution map S is empty, and therefore the predicate interpretation $[\![p^V]\!]S$ evaluates to p^V.

We extend the notion of symbolic execution from paths as defined in Figure 5.3 to programs by considering a function `paths` that, given a program, lazily generates paths through it. At non-deterministic branches, it forks and generates two separate paths for each of the branches. Unlike traditional symbolic execution, we do not specify a predetermined heuristic for selecting which direction to take at branches. Instead we let the symbolic executor generate any feasible paths. Later in Section 5.4.3, we will use particular solutions to guide the symbolic executor through the unknown program.

Theorem 5.1 (No infeasible paths) *The symbolic executor only generates path constraints for feasible paths.*

PROOF: To prove that only feasible paths are explored, it suffices to prove that no path constraint generated through symbolic execution implies `false`. First, notice that the assignment rule only adds an equality for a versioned variable that does not already appear in the path constraint (as we increment its version), and therefore the trace remains feasible if it was before the application of the rule. Second, the premise for assume ensures that the addition of the predicate will not make the path constraint infeasible, and therefore applications of this rule also cannot result in an infeasible path constraint. Notice that the absence of

$$\frac{\tau \wedge [\![p^V]\!]S \not\Rightarrow \texttt{false} \qquad \tau' = \tau \wedge p^V \qquad \tau' \notin \{\tau_i\}}{\texttt{SymEx}^S_{\{\tau_i\}}(\tau, V, \texttt{assume}(p)) \;\; = \;\; \tau', V}$$

$$\frac{v = V[x] + 1 \qquad \tau' = \tau \wedge (x^v = e^V) \qquad \tau' \notin \{\tau_i\}}{\texttt{SymEx}^S_{\{\tau_i\}}(\tau, V, x := e) \;\; = \;\; \tau', V[x \mapsto v]}$$

$$\frac{\texttt{SymEx}^S_{\{\tau_i\}}(\tau, V, st_1) = \tau_1, V_1}{\texttt{SymEx}^S_{\{\tau_i\}}(\tau, V, st_1; st_2) \;\; = \;\; \texttt{SymEx}^S_{\{\tau_i\}}(\tau_1, V_1, st_2)}$$

Figure 5.3: The formalism for the symbolic executor.

a rule for when it does imply `false` ensures that these rules will be stuck for those paths. Lastly, by simple induction, the rule for sequence ensures that if each subsequence of a path is feasible then its combination is feasible. It is easy to verify that the parameters S and $\{\tau_i\}$ do not invalidate the soundness of the symbolic execution.

□

5.4 PINS: Synthesizing programs using symbolic testing

In this section, we describe the steps: safety and termination constraint generation (Section 5.4.1), SMT reduction (Section 5.4.2), and path generation (Section 5.4.3) that make up our PINS algorithm (Section 5.4.4).

5.4.1 Safety and Termination Constraints

We now describe how we approximate safety and termination constraints using symbolic path constraints over $\texttt{Prog}\circ fg$.

Safety constraints using path constraints First, let us consider the task of *verifying* whether a known $\widehat{\texttt{Prog}}$, i.e., an instantiation of $\widehat{\texttt{Prog}}$ with values for its unknowns, satisfies Eq. (5.3), given Prog and spec. One way to approach this problem is to look for approximate guarantees, as in concrete or symbolic execution-based testing, and to ensure that the specification is met on some carefully chosen set of paths through the program. As the set of paths explored becomes larger, the guarantee becomes stronger. To check safety, we can generate path constraints τ over the composed program $\texttt{Prog} \circ \widehat{\texttt{Prog}}$ using the empty solution map $S = \emptyset$, path constraint set $\{\tau_i\} = \emptyset$, and initial version map $V_{\texttt{init}}$:

$$\texttt{SymEx}^S_\emptyset(\texttt{true}, V_{\texttt{init}}, t) = \tau, V \quad \text{where } t \in \texttt{paths}(\texttt{Prog} \circ \widehat{\texttt{Prog}}) \tag{5.4}$$

For each path constraint τ (and version map V) generated above we can check if the safety constraint for the specification holds:

$$\forall X : \tau \Rightarrow \texttt{spec}^V \tag{5.5}$$

where X is the set of all program variables in τ and spec, and we lift the specification to the version map at the end of the path because it specifies a relation at the end.

Notice that in the presence of loops this process will very rarely be complete, as even a single loop can potentially yield an infinite number of unique finite paths. Still, the larger the number of paths checked the better the assurance will be.

The following simple lemma states that symbolic execution is sound and complete with respect to concrete executions:

Lemma 5.1 (Soundness, Completeness of `SymEx`) *There exists an input, i.e., concrete values for $\vec{v_{\mathtt{in}}}$, for which execution of* $\mathtt{Prog} \circ \widehat{\mathtt{Prog}}$ *ends in a state that does not satisfy* `spec`*, if and only if* `SymEx` *generates a path constraint that does not satisfy Eq. 5.5. On the other hand, for all inputs the execution of* $\mathtt{Prog} \circ \widehat{\mathtt{Prog}}$ *ends in a state that satisfies* `spec`*, if and only if every path constraint generated by* `SymEx` *satisfies Eq. 5.5.*

Next, let us consider the task of *synthesizing* values for the unknowns in $\widehat{\mathtt{Prog}}$—or more briefly *synthesizing* $\widehat{\mathtt{Prog}}$—such that it satisfies Eq. (5.3), given `Prog` and `spec`. We assume that we have a flowgraph template *fg* for $\widehat{\mathtt{Prog}}$ that consists of assignments of the form $x := \varepsilon$ (the assigned expression is unknown), and assumes of the form $\mathtt{assume}(\rho)$ (the assumed predicate is also unknown). The path constraint τ generated for some path $t \in \mathtt{paths}(\mathtt{Prog} \circ fg)$ will now have unknown expressions and predicates (lifted to the appropriate versions), and the safety constraint is as before:

$$\mathtt{safepath}((\tau, V), \mathtt{spec}) \doteq \forall X : \tau \Rightarrow \mathtt{spec}^V$$

However, `safepath` is implicitly quantified with $\exists E, K$ where E is the set of all unknown expression symbols ε, and K is the set of all unknown predicate symbols ρ. that appear in τ and spec.

Notice that with this existential over unknowns and universal over program variables, the constraint has exactly the form that a verification condition used for invariant inference has. Typical invariant inference tools, such as those described in Chapters 2 and 3, solve for I from verification conditions of the form $\exists I \forall X : vc$, where I is an unknown invariant. We can therefore borrow techniques (suitably modified to take care of variables version numbers) that are devised for invariant inference and apply them to expression and predicate inference, as we will show in Section 5.4.2. The greater the number of paths for which the above constraint is asserted, the greater the safety ensured.

We can then define the safety constraint for the entire program as:

$$\texttt{safety}(\texttt{Prog} \circ \widehat{\texttt{Prog}}) \doteq \bigwedge_{t \in \texttt{paths}(\texttt{Prog} \circ \widehat{\texttt{Prog}})} \texttt{safepath}(t)$$

where again the constraint is implicitly quantified with $\exists E, K$. Greater assurance can be had by considering more and more conjuncts, each corresponding to a different path t in the program.

Termination constraints using path constraints We now add constraints that ensure termination of the synthesized program. Since loops can be easily identified in the structured programs we consider, we prove each loop terminates by discovering its ranking function, and the entire program terminates if all loops terminate. Our approach for discovering ranking functions is based on assumptions that have been shown reasonable in practice [69, 67, 24]. First, we assume that the loop guard implies an (upper or lower) bound on the ranking function. For example, if $x < y$ is the loop guard then $y-x-1$ is a candidate ranking function (bounded from below by 0 and implied by the loop guard, i.e., $x < y \Rightarrow y-x-1 \geq 0$). Second, we assume that the ranking function, if lower bounded, does not increase in any of the inner loops, and if upper bounded, does not decrease in any of the inner loops. Then we can just check the statements immediately in the body of the loop without worrying about the inner loops modifying its termination argument. The inner loops are proved terminating using their own ranking functions. Consider a loop $l = \texttt{while}(\rho_l)\{B_l\}$ with loop guard ρ_l and body B_l. We assume that the ranking function for a loop l is an unknown expression η_l on which we impose constraints for *boundedness* and *strictly decreasing*, and whose proof may require *dynamic invariants*—in the spirit of trace-based invariant generation tools [102, 103].

Boundedness Under our assumption about the relation of the (unknown) loop guard ρ_l to the ranking function η_l, we impose the following constraints:

$$\texttt{bounded}(l) \doteq \forall X : \rho_l \Rightarrow \eta_l \geq 0$$

Note that here the loop guard and ranking function are *not versioned* and the constraint is implicitly quantified with $\exists \rho_l, \eta_l$.

Strictly decreasing We assume that the inner loops do not affect the termination argument for their enclosing loops[2]. In this case, we can use the path constraints for all paths through B_l—always taking the exit branch for inner loops—to ensure that the ranking function strictly decreases:

$$\texttt{decrease}(l) \doteq \bigwedge_{\tau, V \in exec} \forall X : \tau \Rightarrow {\eta_l}^V < {\eta_l}^0$$

where *exec* is the set of path constraints for all paths through B_l. Notice that we can enumerate *all* paths through B_l because it is necessarily acyclic after we discount the inner loops.

Dynamic Invariants There are cases in which just the path constraint through the body of the loop may not be sufficient to prove that the ranking function decreases in a loop iteration. In these cases, we observe that two additional facts are known when traversing the body of the loop. First, the (unknown) loop guard holds at the entry to the loop, i.e., ρ_l^0 can be assumed in the proof. Second, there exists an (unknown) invariant ϕ_l that can be assumed in the proof, which holds on every path through the loop and holds at the end of every path that leads up to the loop. Incorporating the loop invariant and loop guard into the constraint we get:

$$\texttt{decrease} - \texttt{inv}(l) \doteq \begin{array}{l} \bigwedge_{\tau', V' \in init} \forall X' : \tau' \Rightarrow \phi_l^{V'} \quad \wedge \\ \bigwedge_{\tau, V \in exec} \forall X : \tau \wedge \phi_l^0 \Rightarrow \phi_l^V \quad \wedge \\ \bigwedge_{\tau, V \in exec} \forall X : \tau \wedge \rho_l^0 \wedge \phi_l^0 \Rightarrow {\eta_l}^V < {\eta_l}^0 \end{array}$$

where *exec* are path constraints for paths through the body of the loop as before, while *init* are path constraints for paths leading up to the loop entry. Notice that

[2] If the assumptions do not hold in some case, then because of the particular strategy we use for exploring addition paths (Section 5.4.3), the path exploration will go into an infinite loop, indicating this scenario to the user. In our benchmarks we never encountered this case.

we cannot enumerate all of *init*, so we pick the ones for paths that were explored for the safety constraint.

The termination constraints for the entire unknown program is:

$$\texttt{terminate}(P^{-1}) \doteq \bigwedge_{l \in \texttt{loops(P}^{-1}\texttt{)}} \texttt{decrease}(l) \wedge \texttt{bounded}(l)$$

where again the constraint is implicitly quantified with $\exists E, K$, but in addition also with existentials over ranking functions and invariants, i.e., $\exists \eta_l \phi_l$. The function $\texttt{loops}(\widehat{\texttt{Prog}})$ returns all such syntactically identified loops in $\widehat{\texttt{Prog}}$, which is possible because the program is structured. Notice that again the constraint has the alternating quantification as found in invariant generation constraints.

5.4.2 Satisfiability-based Reduction

We now describe how the safety and termination constraints we generate can be efficiently solved using the techniques developed in Chapter 3. We have noted that the constraints are $\exists\forall$ quantified exactly like the constraints for invariant generation. Tools for verification solve constraints with "there exist" invariant unknown. We use these tools for invariant inference to solve for the "there exists" expressions, predicates and ranking functions. This is similar to our approach in Chapter 4, but different in that now the constraints do not mention invariants at all. Yet, the tools from Chapters 2 and 3 work well for inferring the expressions, predicates and ranking functions that we require. As described in previous chapters, this solving strategy consists of reducing termination and safety constraints to SAT instances that we can solve using off-the-shelf solvers.

We summarize the functionality of the satisfiability-based invariant generation tool, $\mathtt{VS}^3_{\mathtt{PA}}$, we employ. $\mathtt{VS}^3_{\mathtt{PA}}$ takes as input a set of ($\exists\forall$-quantified) constraints *cnstr* and a predicate set Π_p and an expression set Π_e. The key idea in the reduction is to associate with each unknown predicate ρ and $\bar{\rho}$ pair a boolean indicator $b_{\rho\mapsto\bar{\rho}}$ that if assigned `true` indicates that ρ contains the predicate $\bar{\rho}$ and if assigned `false` that it does not. Similar indicators are associated with unknown and known expression pairs. Then the tool makes SMT queries (which for us also need to reason about version numbers) over *cnstr* to generate boolean constraints over the indicators. From the queries it generates a SAT instance, which is then solved using standard SAT solvers. The tool infers subsets, and therefore for each unknown expression ε we assert a constraint to ensure that it maps to singleton sets.

The solution strategy consists of reducing the problem to a SAT instance, and so we can ask it to enumerate solutions to the SAT instance. We use the wrapper

$$\mathtt{satReduceAndSolve}(\mathit{cnstr}, \Pi_p, \Pi_e, m)$$

to denote these calls to $\mathtt{VS}^3_{\mathtt{PA}}$. The parameter m indicates that the wrapper enumerates m solutions (or less if less than m exist), each satisfying *cnstr*. Each of these m solutions provides an assignment of unknowns in *cnstr* to single expressions from Π_e (for unknown expressions) or subsets from Π_p (for unknown predicates).

5.4.3 Directed path exploration using solution maps

We now describe a technique for exploring paths relevant to a particular solution map and directed towards refining the space of solutions for $\widehat{\mathtt{Prog}}$. We first introduce the notion of spurious and valid solution maps:

Definition 5.2 (Spurious and valid solution maps) *We call a solution map S* spurious *if there exists a path $t \in \texttt{paths}(\texttt{Prog} \circ fg)$ whose path constraint τ and version map V are such that $[\![\tau]\!]S \not\Rightarrow \texttt{spec}^V$. If no such path constraint exists then we call the solution* valid*.*

In conjunction with Lemma 5.1, this definition implies that for spurious solution maps S there exist concrete input values for which the execution of $\texttt{Prog}\circ[\![\widehat{\texttt{Prog}}]\!]S$ violates the specification while for valid candidates no such inputs exist.

Note that computing whether a solution S is spurious or valid is in general intractable[3] using symbolic execution, as that may require exploring an infinite number of paths. In the absence of this knowledge suppose we still wanted to explore a new path t (in $\texttt{Prog}\circ\widehat{\texttt{Prog}}$) that would be "relevant" to S, i.e. if S were valid then the constraints generated for t should not exclude S from the space, while if S were spurious then the constraints generated for t should be likely to eliminate S from the space. To describe such relevant paths we need the notion of infeasibility of paths *with respect to S*:

Definition 5.3 ((In)feasibility with respect to a solution map) *A path is* feasible *with respect to a solution map S if $[\![\tau]\!]S \not\Rightarrow \texttt{false}$, where τ is the path constraint corresponding to the path. A path is* infeasible *with respect to the S otherwise.*

[3] Note that here we differ from previous techniques that use formal verifiers [245], as they assume that a verifier exists that classifies solution maps as spurious or valid. They use the counterexample to spurious solution maps to refine the space, or the proof for the valid solutions to terminate. On the other hand, we do not have that luxury.

Now note that a path t' that is infeasible with respect to S will *not* be relevant to S. If t' is infeasible with respect to S then $[\![\tau]\!]S \Rightarrow \texttt{false}$ and the safety constraint corresponding to t' (of the form $\forall X : \tau \Rightarrow \texttt{spec}^V$) is trivially satisfied by S because its antecedent evaluates to `false`. Therefore adding the safety constraint corresponding to t' will never eliminate S (and the class of solutions it represents) from the solution space.

Therefore, our objective is to add new paths such that each solution map satisfies as many path constraints non-trivially as possible. A plausible but impractical approach to generating feasible paths is to randomly add paths from $\texttt{paths}(\texttt{Prog} \circ fg)$. Consider a program and inverse with a nested loop each. Even if we were to consider only 3 unrollings, then for each unrolling of the outer loop the inner loop can be unrolled $0\,..\,3$ times, resulting in $4^0 + 4^1 + 4^2 + 4^3 = 85$ possible paths in each of `Prog` and *fg* and consequently 7225 paths through both. We have found that attempts to refine the space using random exploration does not terminate even for the simplest programs.

Instead our directed path exploration, parametrized by S, constructs paths *feasible* with respect to S. By precluding infeasibility, we force the candidate to satisfy the constraint generated from this new path non-trivially. Consequently, if S is spurious, it is likely that it will fail to satisfy the safety constraint for the new path. If on the other hand, the solution is valid then it will satisfy the new safety and termination constraints by definition (and do so non-trivially). Fortunately, we have the machinery already in place to do this. Instead of running the symbolic executor with an empty solution map, as in Section 5.4.1, we instead run it with the solution map S. This changes the behavior of symbolic execution on assumes with unknown predicates ρ if $\rho \in \texttt{dom}(S)$. In the rule for `assume` in Figure 5.3, instead of checking $\tau \wedge \rho^V \not\Rightarrow \texttt{false}$ the executor will now check $\tau \wedge \bar{\rho}^V \not\Rightarrow \texttt{false}$, where $\bar{\rho} = S[\rho]$. Notice that it is important that we assert termination constraints before attempting to run symbolic execution using S. If termination is not asserted and S corresponds to an infinite loop, then the parametrized symbolic execution will never reach the end of the program. The following theorem holds for parametrized symbolic execution:

Theorem 5.2 *For any path constraint τ that is the output of symbolic execution with solution map S, the path corresponding to τ is feasible with respect to S.*

Our path generation strategy, for the case of valid and spurious solutions, affects the solutions space as follows:

Paths feasible with respect to valid solutions Let S be a valid solution map and let t be a path that satisfies Theorem 5.2 with with respect to S. Then the constraints from t will *not* eliminate S because S, being valid, by definition satisfies the specification on all paths. On the other hand, the constraints may eliminate other spurious solutions.

Paths feasible with respect to spurious solutions A path that satisfies Theorem 5.2, is only guaranteed to be feasible with respect to S, which may be spurious. It is important to assert the corresponding constraint because it is likely to eliminate the spurious S (and other solutions that are similar to it) despite the fact that

```
Input: Original program Prog,
       Specification spec,
       Number of solutions from SAT solver $m$.
Output: Inverted Program $P^{-1}$ or "No Solution".
begin
    $fg := \overline{\texttt{fg}}(\texttt{Prog})$; $preds := \overline{\texttt{preds}}(\texttt{Prog})$; $exprs := \overline{\texttt{exprs}}(\texttt{Prog})$;
    $prog := \texttt{Prog} \circ fg$;
    $pc := \texttt{SymEx}^{\emptyset}_{\emptyset}(\texttt{true}, V_{\texttt{init}}, t)$; with $t \in \texttt{paths}(prog)$;
    $pcset := \{pc\}$;
    $cnstr := \texttt{terminate}(fg)$;
    while $(*)$ do
        $cnstr := cnstr \wedge \texttt{safepath}(pc, \texttt{spec})$;
        $sols := \texttt{satReduceAndSolve}(cnstr, preds, exprs, m)$;
        if $sols = \emptyset$ then
            return "No Solution"; /* Refine abstraction */
        if $\texttt{stabilized}(sols)$ then
            return $[\![fg]\!]sols[0]$;
        $S := \texttt{pickOne}(sols)$;
        $pc := \texttt{SymEx}^{S}_{pcset}(\texttt{true}, V_{\texttt{init}}, t)$; with $t \in \texttt{paths}(prog)$;
        $pcset := pcset \cup \{pc\}$;
end
```

Figure 5.4: The PINS semi-algorithm.

the path may be infeasible for every other valid solution S'. This is in contrast to a traditional symbolic executor where infeasible paths only add overhead. With unknown expressions and predicates, paths that are feasible with respect to spurious solutions (but may be infeasible with respect to valid solutions) yield constraints that are likely to eliminate the spurious solutions and are therefore useful.

5.4.4 PINS: Path-based Inductive Synthesis

Figure 5.4 shows our iterative path-based synthesis algorithm. We mine from the original input program Prog a flowgraph template $\overline{\texttt{fg}}(\texttt{Prog})$ and predicate and expression sets $\overline{\texttt{preds}}(\texttt{Prog})$ and $\overline{\texttt{exprs}}(\texttt{Prog})$ (Section 5.5.1). We compose the flowgraph template with Prog to get the program *prog* over which we run the symbolic executor. The symbolic executor explores some path t and generates the corresponding path constraint pc, which we log in the set of explored paths $pcset$. We maintain a constraint $cnstr$ that we initialize to the termination constraint and then in each iteration add an additional safety constraint from `safepath`. In each iteration we query the SAT solver for solutions to the current constraint $cnstr$ under the predicate and expression sets mined earlier and ask for m solution maps. We pick one of those solutions using `pickOne`, which returns the solution with the fewest feasible paths with respect to it.

`pickOne` ensures that we preferentially add paths for solutions that currently have fewer feasible paths. This process prunes the space by ensuring that only those solutions remain that satisfy the specification over many paths. Typically, we have found that the algorithm converges to the valid solutions in a few iterations.

The algorithm terminates when `stabilized` holds. The choice of this function depends on the guarantees required from `PINS`, and we omit a precise definition here to permit flexibility. In our implementation, we terminate when only one solution remains. Alternatively, we can imagine terminating whenever the set of candidates has fewer than m elements and then use other, more lightweight mechanisms (e.g., concrete testing) to eliminate any remaining spurious solutions.

Notice that the constraints generated are implicitly existentially quantified at the outermost level as $\exists E, K, \{\eta_l\}_l$. The constraint solving technique assigns appropriate known values to these from the given expression and predicate sets (with the candidates for ranking functions constructed from the predicates).

Runs of `PINS` We now describe the result of running `PINS` when the inverse exists and when it does not exist under the given fg, Π_p, and Π_e. When the inverse does not exist then `PINS` eventually finds paths whose constraints are unsatisfiable, and therefore the solution set is empty. At that point, the user is asked to modify the input. If, on the other hand an inverse exists, then `PINS` finds paths such that the solution space only contains valid inverses. In such cases it degenerates to a symbolic execution-based verifier, continuously attempting to find paths to narrow the search space further, but failing to eliminate any of the valid solutions, until terminated by the `stabilized` function.

Notice that the algorithm ensures (using *pcset*) that no paths are revisited. On the other hand additional paths are only added if they are feasible with respect to a given solution S. Consider the case where no paths exist that are feasible with respect to S but whose path constraint is not already in *pcset*. In this particular case, we have exhaustively validated that S satisfies the specification on all paths through the program.

5.5 `PINS` in practice

In this section we describe our approach to mining the template (Section 5.5.1) and our support for axioms (Sections 5.5.2) and recursion (Section 5.5.3) We then describe how we instantiate `PINS` for sequential and parallel composition to handle inversion (Section 5.5.4) and client-server synthesis (Section 5.5.5), respectively.

5.5.1 Mining the flowgraph, predicates and expressions

In this section, we describe how to mine the *flowgraph* template, *expression* and *predicate* sets used in the `PINS` algorithm (Figure 5.4). It is most convenient to consider `Prog` written in a language that makes the structured control flow explicit:

$$\begin{array}{rcl} K & ::= & x := \bar{\varepsilon} \;\mid\; {}^{a}F \;\mid\; K; K \\ F & ::= & \texttt{if}(\bar{\rho})\; K \;\texttt{else}\; K \;\mid\; \texttt{while}(\bar{\rho})\; K \;\mid\; \texttt{main}\; K \\ a & ::= & \downarrow \mid \uparrow \mid \times \end{array}$$

The language consists of sequences of known statements K and structured control flow elements F that the user annotates with *tags* a. The annotation is either a forward $\downarrow$, a backwards $\uparrow$, or a deletion $\times$ tag. Tags indicate the direction of statements *in the inverse* P^{-1} with respect to the original program.

Note that, ignoring the tags a, a program in the language K can be translated to the language *stmt* in a standard manner. To translate $\texttt{if}(\bar{\rho})\; K_1\; K_2$, we output a non-deterministic branch with $\texttt{assume}(\bar{\rho})$ followed by the translation for K_1 and $\texttt{assume}(\neg\bar{\rho})$ followed by the translation for K_2. To translate $\texttt{while}(\bar{\rho})\; K_1$, we output a non-deterministic branch with $\texttt{assume}(\bar{\rho})$ followed by the translation for K_1 and going back to the loop on one branch, and $\texttt{assume}(\neg\bar{\rho})$ on the other. The only non-standard construct is `main`, which we use to indicate the entry point of the program. The presence of `main` allows us to associate a tag with the outermost set of statements.

By allowing the user to specify the $\downarrow, \uparrow$, or $\times$ tag, we provide the user with the flexibility to influence the template mining, while by limiting it to one tag at the head of each control flow structure, we minimize the annotation burden.

Given a tagged program Prog in the language K, we define functions $\overline{\texttt{fg}}$, $\overline{\texttt{pred}}$, and $\overline{\texttt{expr}}$ that mine using structural induction the flowgraph, predicate set, and expression set for the inverse P^{-1}. The key idea behind $\overline{\texttt{fg}}$ is to translate an assignment $x := \bar{\varepsilon}$ to either $x := \varepsilon_0$ (if the tag is $\downarrow$) or to $\langle v_1, v_2 \ldots := \varepsilon_1, \varepsilon_2 \ldots \rangle$ (if the tag $\uparrow$), where $v_1, v_2 \ldots \in \texttt{vars}(\bar{\varepsilon})$ and ε_i are fresh unknown expression symbols. $\overline{\texttt{pred}}$ recursively extracts predicate guards from the original program and also generates predicates from some commonly occurring patterns in program-inverse pairs. $\overline{\texttt{expr}}$ also recursively extracts expressions from the original program, but applies a heuristic expression inverter, converting $-$ to $+$ etc., when the tag is $\uparrow$ and returns the expressions as is when the tag is $\downarrow$.

The functions $\overline{\texttt{fg}}$, $\overline{\texttt{pred}}$, and $\overline{\texttt{expr}}$ are just the corresponding ones shown in Figure 5.5 with a postprocessing step that renames variables so that the variables of Prog do not interfere with the variables of P^{-1}. The renaming is assumed to be consistent, e.g., v is always renamed to to v'. This is required because our technique for synthesis composes the two programs together, and we do not want extraneous values at the end of the first program to flow into the second program. Renaming ensures that this does not happen.

5.5.2 Axiomatization for handling Abstract Data Types

A major concern for modular synthesis is proper handling of abstract data types (ADTs). A key feature of our symbolic executor, and consequently of PINS,

$$\texttt{fg}^{a}(x := \bar{\varepsilon}) = \begin{cases} \{x := \varepsilon\} & a = \downarrow \\ \langle v_1, v_2 \ldots\rangle := \langle \varepsilon_1, \varepsilon_2, \ldots\rangle \forall v_i \in \texttt{vars}(\bar{\varepsilon}) & a = \uparrow \end{cases}$$

$$\texttt{fg}^{a}(K_1; K_2) = \begin{cases} \texttt{fg}^{a}(K_1); \texttt{fg}^{a}(K_2) & a = \downarrow \\ \texttt{fg}^{a}(K_2); \texttt{fg}^{a}(K_1) & a = \uparrow \end{cases}$$

$$\texttt{fg}({}^{a}\texttt{if}(\bar{\rho})\ K_1\ \texttt{else}\ K_2) = \begin{cases} \texttt{if}(\rho)\ \texttt{fg}^{a}(K_1)\ \texttt{else}\ \texttt{fg}^{a}(K_2) & a \neq \times \\ \texttt{skip} & \text{otherwise} \end{cases}$$

$$\texttt{fg}({}^{a}\texttt{while}(\bar{\rho})\ K_1) = \begin{cases} \texttt{while}(\rho)\ \texttt{fg}^{a}(K_1) & a \neq \times \\ \texttt{fg}^{a'}(K_1) & \text{otherwise } (a'\text{: annotation} \\ & \quad \text{on the enclosing block)} \end{cases}$$

$$\texttt{fg}({}^{a}\texttt{main}\ K_1) = \texttt{main}\ \texttt{fg}^{a}(K_1) \quad a \neq \times$$

$$\texttt{preds}(x := \bar{\varepsilon}) = \emptyset$$

$$\texttt{preds}(K_1; K_2) = \texttt{preds}(K_1) \cup \texttt{preds}(K_2)$$

$$\texttt{preds}({}^{a}\texttt{if}(\bar{\rho})\ K_1\ \texttt{else}\ K_2) = \begin{cases} \{\bar{\rho}\} \cup \texttt{preds}(K_1) & a \neq \times \\ \qquad \cup\, \texttt{preds}(K_2) & \\ \emptyset & \text{otherwise} \end{cases}$$

$$\texttt{preds}({}^{a}\texttt{while}(\bar{\rho})\ K_1) = \begin{cases} \{\bar{\rho}\} \cup \texttt{preds}(K_1) & a \neq \times \\ \emptyset & \text{otherwise} \end{cases}$$

$$\texttt{preds}({}^{a}\texttt{main}\ K_1) = \begin{cases} \texttt{preds}(K_1) & a \neq \times \\ \emptyset & \text{otherwise} \end{cases}$$

$$\texttt{expr}^{\downarrow}(x := \bar{\varepsilon}) = \{\bar{\varepsilon}\}$$

$$\texttt{expr}^{\uparrow}(x := \bar{\varepsilon}) = \{\texttt{invop}(\bar{\varepsilon})\}$$

$$\texttt{expr}^{a}(K_1; K_2) = \texttt{expr}^{a}(K_1) \cup \texttt{expr}^{a}(K_2) \quad a \neq \times$$

$$\texttt{expr}({}^{a}\texttt{if}(\bar{\rho})\ K_1\ \texttt{else}\ K_2) = \begin{cases} \texttt{expr}^{a}(K_1) \cup \texttt{expr}^{a}(K_2) & a \neq \times \\ \emptyset & \text{otherwise} \end{cases}$$

$$\texttt{expr}({}^{a}\texttt{while}(\bar{\rho})\ K_1) = \begin{cases} \texttt{expr}^{a}(K_1) & a \neq \times \\ \emptyset & \text{otherwise} \end{cases}$$

$$\texttt{expr}({}^{a}\texttt{main}\ K_1) = \begin{cases} \texttt{expr}^{a}(K_1) & a \neq \times \\ \emptyset & \text{otherwise} \end{cases}$$

$$\texttt{invop}(x) = x$$

$$\texttt{invop}(f) = g \quad \text{where } (f, g), (g, f) \in \{(+, -), (*, /)\}$$

$$\texttt{invop}(f \cdot g) = \texttt{invop}(g) \cdot \texttt{invop}(f)$$

$$\texttt{invop}(\texttt{upd}(A, i, fn(\texttt{sel}(B, j)))) = \texttt{upd}(B, j, \texttt{invop}(fn)(\texttt{sel}(A, i)))$$

Figure 5.5: Automatically mining flowgraphs, predicate and expression sets.

$$\frac{\begin{array}{c}\vec{V_r} = \vec{V}[i+1 \mapsto V_{\texttt{init}}(i+1)] \\ \tau_{\texttt{entry}} = (\vec{v_{\texttt{in}}}^{(i+1,0)} = \vec{v_{\texttt{args}}}^{\vec{V}[i]}) \qquad \texttt{SymEx}^{S}_{\{\tau_i\}}(\tau \wedge \tau_{\texttt{entry}}, \vec{V_r}, i+1, \texttt{Prog}) = \tau', \vec{V}' \\ \vec{V}'' = \vec{V}[i][v \mapsto \vec{V}[i][v] + 1] \quad \forall v \in \vec{v_{\texttt{ret}}} \qquad \tau_{\texttt{exit}} = (\vec{v_{\texttt{ret}}}^{\vec{V}''[i]} = \vec{v_{\texttt{out}}}^{\vec{V}'[i+1]})\end{array}}{\texttt{SymEx}^{S}_{\{\tau_i\}}(\tau, \vec{V}, i, \vec{v_{\texttt{ret}}} := \texttt{rec}(\vec{v_{\texttt{args}}})) \quad = \quad \tau' \wedge \tau_{\texttt{exit}}, \vec{V}''}$$

Figure 5.6: Handling recursion in PINS. $V_{\texttt{init}}(k)$ indicates the initial version map for stack depth k and is maps all variables in Prog to the version number $(k, 0)$. Also, we use the notation $(k, j) + 1$ to denote $(k, j + 1)$.

is its extensibility by means of axioms that is borrows from the use of $\texttt{VS}^3_{\texttt{AX}}$ (which builds on top of $\texttt{VS}^3_{\texttt{PA}}$) from Chapter 4. For an ADT, we assert quantified axioms about its interface functions in the SMT solver. For instance, consider the String ADT. Suppose a program uses its interface functions `append`, `strlen`, and `empty`. Then, among others, we assert the following in the SMT solver:

$$\begin{aligned} &\texttt{strlen}(\texttt{empty}()) = 0 \\ \forall x, y\text{: } &\texttt{strlen}(\texttt{append}(x, y)) = \texttt{strlen}(x) + \texttt{strlen}(y) \\ \forall x, c\text{: } &\texttt{strlen}(\texttt{append}(x, \text{`}c\text{'})) = \texttt{strlen}(x) + 1 \end{aligned}$$

We employ this facility to reason about operations that are difficult for SMT solvers. For instance, we assert an axiom $\forall x \neq 0 : x \times (1/x) = 1$ because reasoning about multiplication and division in general is hard for SMT solvers. Additionally, we will use this in the next section to enforce that for communicating programs composed in parallel, each message send is matched with a corresponding receive.

5.5.3 Recursion

To handle recursive calls, we augment our language with the statement $\vec{v_{\texttt{ret}}} :=$ $\texttt{rec}(\vec{v_{\texttt{args}}})$[4]. Additionally, the symbolic executor now maintains a *stack* of version maps $\vec{V}$ and a current stack depth i. The top of the stack $\vec{V}[i]$ contains the version map for the current stack depth. Also, variables now have versions that are tuples (d, i), where d denotes the stack depth and i denotes the version number at that depth.

We add a symbolic execution rule, shown in Figure 5.6, to interpret the recursive call. The rule, for a recursive call at depth i, pushes on the stack an initial version map $V_{\texttt{init}}(i+1)$. The initial version map $V_{\texttt{init}}(i+1)$ is a map that assignes all variables to default initial version 0 at stack depth $i+1$. It then runs the symbolic executor over the program `Prog` corresponding to the recursion, with an initial trace $\tau \wedge \tau_{\texttt{entry}}$, the new stack of version maps, and the stack depth, to yield the output path constraint τ'. (We ignore the stack of version maps $\vec{V}'$ that results after the recursion bottoms out because the local variables go out of scope then.) $\tau_{\texttt{entry}}$ and $\tau_{\texttt{exit}}$ take care of the passing the function arguments and return values by asserting appropriate equalities between variables (arguments and formal parameters; return values and assigned variables) at different stack depths. Lastly, because they are assigned to, the versions of variables getting the return values (at depth i) are incremented.

An almost identical rule suffices for handling arbitrary procedure calls. While

[4] This construct does not allow mutually recursive functions, but it can trivially be extended.

straight-forward, we have not yet experimented with arbitrary interprocedural synthesis.

5.5.4 Sequential composition: Synthesizing Inverses

In our modeling of sequential composition, we instantiate the compose operator "$\circ$" as the sequencing operator " ; ". We may need to be careful about ensuring that there is only one exit point to the first program.

For program inversion, we consider concatenating the original program followed by the template unknown inverse. This leads to a definition of inversion that is mathematically similar to that of a left inverse of a function. If we concatenate the template unknown inverse followed by the inverse, we will generate right inverses. In fact, for efficiency, for the case that both the left and right inverse exist and are the same for a particular program—which is the case for paired computation, such as compression, formatting etc.—we can even generate paths over both types of concatenation. For the experiments in this chapter, though, we generate inverses that technically are left inverses.

5.5.5 Parallel composition: Synthesizing Network Programs

In our modeling of parallel composition, we instantiate the compose operator "$\circ$" as the operator " $||$ ". For us, programs composed in parallel run simultaneously while only interacting through message passing. We augment our symbolic executor in two ways to handle parallel composition. First, under the assumption that the composed programs do not share common variables, we define $\texttt{paths}(\texttt{Prog}_1 \mid\mid \texttt{Prog}_2)$ as $\{t_1; t_2 \mid t_1 \in \texttt{paths}(\texttt{Prog}_1), t_2 \in \texttt{paths}(t_2)\}$. Notice that in contrast to the traditional approach of interleaving the executions of programs, we concatenate the path constraints, and we leave it up to the axiomatization of message passing primitives to generate appropriate equalities that connect the two path constraints. This is sound because we assume that the programs do not share variables. Second, we add the premise $\tau \wedge (x^v = e^V) \not\Rightarrow \texttt{false}$ to the rule for handling assignments. In the case of parallel composition, in addition to assumptions leading to infeasibility, assignments may do so too when messages are received that result in additional facts being generated.

Logical clocks for ensuring in-order communication Our approach to modeling communication under parallel composition is inspired by the notion of logical clocks by Lamport [180]. Lamport's clocks ensure that for two distributed processes A and B with two event a and b such that a "happens-before", notated as $a \rightarrow b$, it is the case that $C_A(a) < C_B(b)$. If this consistency constraint is maintained then a total ordering can be imposed on the events of the system.

We ensure such distributed consistency by maintaining logical clocks at each node and updating them, as in Lamport's proposal. For each communicating entity, we associate a clock variable clk. This logical clock is incremented every time the entity sends or receives a message. The increment is encoded as part of the axiom that matches up a send with a receive, and which may cause a buffer equality to be generated, as we show later. (The variable clk is a proof term, and no program statement exists that can manually update the clock.) At the end of each path, in addition to asserting the specification, we now additionally assert that the logical clocks of all entities are equal—ensuring that only in-order communication is allowed and that each send has a corresponding receive and vice versa.

Axioms for buffer equality on message sends and receives We assert *buffer equality axioms* that generate an equality between the message buffer sent and the buffer received. This allows the system to synthesize statements that call the send and receive functions without worrying about their communicating semantics, as this reasoning gets integrated into the SMT solver through the axioms.

Example 5.1 *Consider programs that use uninterpreted functions* `send` *and* `recv` *for communication. An axiom that relates* `send` *and* `recv` *could be the following (essentially providing an abstract semantics for the communication):*

$$\forall \begin{matrix} x, y, y' \\ clk_1, clk'_1, \\ clk_2, clk'_2 \end{matrix} : \begin{pmatrix} clk_1 = clk_2 \wedge \\ (y', clk'_1) = \mathtt{send}(x, clk_1) \wedge \\ (y, clk'_2) = \mathtt{recv}(clk_2) \end{pmatrix} \Rightarrow \begin{pmatrix} clk'_1 = clk_1 + 1 \wedge \\ clk'_2 = clk_2 + 1 \wedge \\ x = y \end{pmatrix}$$

Consider a known client program

$$\mathtt{in}(v); (v, clk_c) := \mathtt{send}(v, clk_c); (v', clk_c) := \mathtt{recv}(clk_c);$$

with postcondition $v' = v+1$. Suppose we wish to synthesize the corresponding (echo-increment) server with the template $(n, clk_s) := \varepsilon_1; (n', clk_s) := \varepsilon_2$. (We generate such templates by augmenting each assignment in the original mined template $n := \varepsilon'_1; n' := \varepsilon'_2$ to simultaneously update the clock variable as well.)

The logical clock preconditions $clk_s = clk_c = 0$ and postcondition $clk_s = clk_c$ are asserted automatically by the system at the start and end of each path, respectively. Then, given the above buffer equality axiom, the only solution that satisfies the clock postcondition and $v' = v + 1$ for the composed program is $\{\varepsilon_1 \mapsto$ $\texttt{recv}(clk_s), \varepsilon_2 \mapsto \texttt{send}(n + 1, clk_s)\}$.

Notice how the use of logical clocks and axioms for buffer equality allows us to seamlessly deal with the problem of synthesis. Logical clocks ensure that no out-of-order communication is possible (soundness) and ensuring buffer equality on message transfers allows us to reason across the communicating entities (completeness). By encoding communication this way we can synthesize communicating programs using our algorithm from before.

5.6 Experiments

We implemented a symbolic executor based directly on rules in Figure 5.3 and 5.6, and used it to implement the `PINS` algorithm (Figure 5.4).

Number of solutions (m) and prioritizing them (`pickOne`*)* `PINS` (Figure 5.4) is parametrized by the number of solutions m and `pickOne`. We use the SAT solver to enumerate m solutions in each step. The objective is to get a fair sampling of solutions on which we apply our prioritization heuristic, while at the same time not spending too much time in the solver. The extremes $m = 1$ and $m = \infty$ are therefore undesirable: $m = 1$ does not allow us to compare solutions, while $m = \infty$

wastes too much time in the SAT solver. In our experiments we chose $m = 10$, which worked well.

Second, we prioritize the m solutions according to a heuristic `pickOne`. Our approach is again based on the observation that spurious solutions typically satisfy the safety and termination constraints by ensuring (through suitable assignments of the unknowns) that a large fractions of the paths are infeasible. Our implemented `pickOne` first counts the number of infeasible paths for each solution map S:

$$\texttt{infeasible}(S) = |\{\tau \mid \tau, V \in \mathit{pcset} \wedge [\![\tau]\!]S \Rightarrow \texttt{false}\}|$$

and then picks a solution map S with a high `infeasible`(S) value. We experimentally validated this heuristic for `pickOne` against another that randomly picks any solution from the m available. In our experiments random selection yields runtimes that are 20% more than with `infeasible`, and therefore we use `infeasible`. We did observe that random selection is better in the initial few iterations but then takes longer to identify the paths that eliminate the last few solutions, as expected, and therefore takes more time overall. This suggests that the ideal strategy would be a hybrid that starts with random selection and then switches to the `infeasible` metric when the number of solutions is small.

The process of synthesis using `PINS` The user annotates the conditionals, loops, and main entry point, with the tags appropriately as described in Section 5.5.1. From the annotated program, we extract the flowgraph, predicate, and expression sets, using the functions shown in Figure 5.5. Currently, we run the extraction functions by hand, but they are trivial to automate. When the synthesis attempt fails for the initial mined values, we modify them suitably using the paths that PINS explores. We will report the number of such changes that we had to do for our experiments later. Our path-based approach is very helpful in identifying the cause of imprecision/inaccuracy in the predicates, expressions, and flowgraph template. On the one hand, if a valid inverse does not exist in our template, then PINS generates paths that eliminate *all* solutions. In this case, we examine the paths and change either the predicates, expressions, or flowgraph. This is very similar to abstraction refinement in CEGAR [148], and therefore we expect the process can be made completely automatic. On the other hand, if a valid inverse does exist in our template, then PINS eliminates all but the valid ones. At that point, we manually inspect each synthesized inverse to confirm that it indeed is valid.

5.6.1 Benchmarks

We synthesized programs in two categories: program inversion, illustrating our technique for sequential composition, and client-server synthesis, illustrating our technique for parallel composition.

5.6.1.1 Program Inversion: Sequential Composition

To demonstrate the feasibility of synthesizing programs that are sequentially composed, we consider three different synthesis tasks: decoders for compression programs, finding the inverses for format conversion programs, and finding the inverses for arithmetic programs.

For all the benchmarks in these domains the specification `spec` is identity, i.e., for all variables v with primitive types we assert $v = v'$, and for all variables A with aggregate types (e.g., arrays, strings) with bounds n we assert $n = n' \wedge \forall 0 \leq i < n \Rightarrow A[i] = A'[i]$, where the primed variables are those at the end of the execution.

Compressors Our first compression benchmark is *run length encoding*, which scans the input for sequences of consecutive characters and outputs characters and their counts (a more complex variant of Figure 5.1(a)). The decoder, which we synthesize, expands each (character, count) pair into the original sequence. Though simple, this example illustrates the need to provide the flexibility of annotating control structures with directions. Of the three structures in the program (the entry point and two loops that are nested), the outer loop is annotated as ↓ to allow processing the stream in the direction it was encoded.

Our second compression benchmark is the *LZ77 encoding* algorithm [272], which is the basis of the popular Deflate algorithm used in `gzip`, `zip`, and `PNG` images. LZ77 compresses data by outputting pointers to identical sequences seen in the past. Therefore, an entry in the output may indicate "copy 5 characters starting from 9 characters behind this point," or more interestingly "copy 7 characters starting from 1 character behind this point." The algorithm takes care of bootstrapping the process by outputting the next unmatched character along with each (pointer, count) pair. The decoder, which we synthesize, reconstructs the original stream from the (pointer, count) pairs and the characters in the encoded stream. Again, we annotated the outer loop with a ↓ tag, and one of the two inner loops with a ×

tag. This was easy to guess since the loop searches for the best match in the input stream, which the decoder would not need to do.

Our last compression benchmark in this category is the *Lempel-Ziv-Welch (LZW) encoding* [265], which is the basis of GIF compression. The algorithm builds an online dictionary using the input data and outputs dictionary indices. The bootstrapping process implicit in the encoder leads to a corner case when the encoder adds a dictionary entry and then immediately outputs its index [265]. The decoder builds an dictionary identical to what the encoder constructed earlier to reconstruct the original stream. The decoder is tricky because the corner case that requires it to construct the next dictionary entry and the output string simultaneously. Our technique automatically synthesizes the decoder with this corner case after we annotate some of the inner control structures with ×. As in LZ77, the inner loop searches for the longest dictionary sequence, which the decoder need not do, so guessing the × was easy.

Formatters Our first formatter benchmark is a program for Base64 MIME encoding that converts its binary input to base 64 encoding with ASCII characters that are both common and printable. We synthesize the inverse program that converts the ASCII printable characters to the original binary stream. The encoder has a nontrivial control structure with an outer loop containing a sequence of two inner loops. We synthesize the inverse using a ↓ tag on each loop.

Our second formatter benchmark is a program for UUEncode binary-to-text encoding that outputs four printable characters for every three bytes of input and adds a header and footer to the output. We synthesize the inverse program to convert the printable text back to the original binary stream.

Our third formatter benchmark wraps data into a variable length packet data format by adding a preamble (the length of the field) to the data bytes for the field. We synthesize the inverse program, which reads the length and appropriate bytes of the data fields to reconstruct the input data.

Our fourth formatter benchmark is a recursive function that takes a description of objects and writes out their XML representation, e.g., for serialization. We synthesize the inverse program, which reads the XML representation and reconstructs the object.

Arithmetic Our first arithmetic benchmark is a simple iterative computation of $\sum i$ that in the ith iteration adds i to the sum. This is another program where it may not be feasible to derive the inverse just by reading the program backwards. In this case, reading backwards one would need to solve for n from $n(n+1)/2$, i.e., solve a quadratic, which is hard to automate. Using a $\downarrow$ tag for the loop, our tool automatically synthesized the inverse that in the ith iteration subtracts i from the sum until it reaches 0.

Our next three arithmetic benchmarks are *vector manipulation* programs for shifting, scaling, and rotating a set of points on the Euclidean plane. These primitives are used frequently in graphics programming and image manipulation. For each operation we synthesize the corresponding inverse.

Our fifth arithmetic benchmark is *Dijkstra's permutation* program from his original note on program inversion [89]. He considered a program that, given a permutation π, computes for the ith element of π the number of elements between $0 \mathinner{\ldotp\ldotp} i$ that are less that $\pi(i)$. The inverse program computes the permutation from an array of these counts. Dijkstra manually derived it from the original program, while we automatically synthesize the inverse.

Our last arithmetic benchmark is a program for in-place computation of the *LU-decomposition* of a matrix using the Doolittle algorithm [223]. The inverse, which has been manually derived before [54] and which we synthesize automatically, is a program that multiplies the lower triangular and upper triangular matrices in-place.

5.6.1.2 Client-Server: Parallel Composition

To demonstrate the feasibility of synthesizing programs that are composed in parallel, we synthesize the client functions from the corresponding functions in a Trivial File Transport Protocol (TFTP) server. We use an open source implementation of a TFTP server as the starting point. The send functions have retry mechanisms to account for network errors with no corresponding code when receiving and therefore we abstract them out into macros.

For all functions we synthesize here, we assert a given specification `spec` for the parallel combination, typically that values (files, counters, data buffers, etc.) on the server end up in corresponding variables on the client or vice-versa. Additionally, we assert that the logical clocks (Section 5.5.5) on both the client and server are identical at the end of the execution. This ensures that all send and receive functions were paired up, and in the right order.

The first function in the server is the *main body*[5] which picks whether the transfer mode is from the server to the client or the other way around, i.e., the command is "get file" or "put file." It then calls the appropriate transfer functions, reading or writing to disk as required. We synthesize the corresponding client function using a ↓ tag on the entry point and ↑ on an inner block.

The second function in the server takes a file and sends it out into packets or reads packets and outputs a file. We synthesize the corresponding inverse using a ↑ tag on the loop for the transfer.

The third set of functions send or get an acknowledgment or a data packet. We synthesize the corresponding client functions using ↑ tags.

The last function in the server takes the fields for acknowledgment or data and wraps it into a packet and sends it. We synthesize the corresponding client function using a ↓ tag.

5.6.2 Experience and Performance

Table 5.1 shows the result of running PINS over our benchmarks. For each benchmark, we present numbers for three aspects of the experiment (1) the benchmark characteristics, (2) the runs of our mining heuristic, and, (3) the runs of PINS. For the benchmark characteristics we list the lines of code and the number of axioms about the uninterpreted functions used in the program. For the runs of our

[5] We simplify the main body of the server by only considering one client accept instead of the infinite loop, so that it corresponds to one client that we are interested in synthesizing.

								Percentage of total time					
Benchmark	LoC	Mined			Total	Num.	Num.	Sym.	SMT	SAT		Total	SAT
		$\|fg\|$	$\|\Pi_p\|$	$\|\Pi_e\|$	Chngs	Axms	Iter.	Exe.	Red.	Sol.	`pickOne`	Time (s)	Size
Run length	12	10	2	8	2	0	7	45%	45%	7%	3%	26.19	668
LZ77	20	13	2	8	4	0	$6^{\dagger}$	98%	1%	<0.1%	<0.1%	1810.31	330
LZW	25	20	2	12	8	15	$4^{\dagger}$	68%	29%	<1%	3%	150.42	373
Base64	22	16	3	6	1	3	$12^{\ddagger}$	42%	57%	<1%	<1%	1376.82	598
UUEncode	12	11	2	6	6	3	7	84%	12%	1%	3%	34.00	177
Pkt Wrapper	10	16	1	6	5	2	6	1%	96%	3%	<1%	132.32	2161
XML Serialize	8	8	0	5	0	6	14	92%	7%	<1%	<1%	55.33	69
$\sum i$	5	5	4	3	2	0	4	50%	38%	4%	8%	1.07	51
Vector shift	8	7	1	6	0	0	3	21%	73%	2%	4%	4.20	187
Vector scale	8	7	1	6	0	1	3	21%	73%	2%	4%	4.41	191
Vector rotate	8	7	1	6	2	1	3	6%	93%	<1%	<1%	39.51	327
Dijkstra's permute	11	10	2	5	6	0	1	96%	2%	<1%	2%	8.44	4
LU-decomp-mul	11	12	3	7	7	2	1	88%	11%	<0.1%	1%	160.24	10
CMD loop	20	13	1	9	2	5	3	15%	80%	<1%	4%	22.10	237
File get-send	14	8	1	7	2	5	1	<1%	>99%	<1%	<0.1%	519.67	3157
Ack get-send	12	4	1	5	0	3	1	5%	89%	1%	4%	1.41	80
Data get-send	7	9	1	5	1	3	1	<0.1%	>99%	<1%	<0.1%	442.29	3920
Pkt get-send	9	5	0	5	0	3	1	17%	72%	2%	9%	1.03	41

Table 5.1: The experimental case studies for `PINS`. Unless indicated otherwise, we run the algorithm (Figure 5.4) until only valid solutions remains (and are not refuted in a subsequent iteration). A superscript of † and ‡ indicate that the solution set contains 2 and 4 remaining solutions, respectively.

mining heuristic we report the sizes of the flowgraph in lines of code, the sizes of the expression and predicate sets, and the total changes that the user had to make to those mined templates. For the runs of PINS, we report the number of iterations it took for the algorithm to converge to a stable set (mostly just one valid inverse that we inspected to be correct). Then we report the fraction of the total time spent in each of the four subparts of the algorithm (symbolic execution, SMT reduction to SAT and SAT solving, and prioritization, i.e., pickOne) and the total time taken in seconds to stabilize and generate the inverse. Lastly, we report the size of the total SAT instance that constrains the system to the stabilized set.

There were three programs in which the stabilized set contained more than one solution before our tool exhausted memory or time. In LZW and LZ77 the solution set contained two solutions each, and for LZW both were valid. There were four solutions left for Base64. For both LZ77 and Base64 only one solution was valid while the rest spurious but the tool ran out of time trying to add new paths.

Our mining heuristics were very accurate in inferring the right flowgraphs, expression, and predicate sets. Of the total 328 non-trivial lines that the user would have had to guess otherwise, our heuristics reduces the burden to modifications in 48 of those, i.e., 15%. Typically these were very simple, e.g., changing '+' to '-' or swapping variables in an expression. Additionally, with the path available from PINS, the modifications were very trivial to infer. We note that PINS stabilizes and synthesizes the inverse for these realistic programs in a few iterations (under 14 at most, and with a median of 3) within very reasonable time (under 30 minutes at most, and with a median of 40 seconds), and the entire SAT constraint is concise and small (at most 3k clauses). We also see from the fraction of time spent in each subpart that symbolic execution and constraint reduction take the most time. Therefore improvements to these will automatically benefit our synthesis technique.

5.7 Summary

In this chapter, we presented `PINS`, an approach to program synthesis that uses symbolic execution to approximate the correctness constraints and satisfiability-based tools, from the previous chapters, to solve them. We applied the technique to program inversion and client-server synthesis. We showed that `PINS` can successfully synthesize a wide variety of realistic programs.

5.8 Further Reading

Inductive and Deductive Synthesis Program synthesis techniques can be classified as belonging to a spectrum that stretches from inductive synthesis on the one end and deductive synthesis on the other. Inductive synthesis is an approach that generalizes from finite instances to yield an infinite state program. One example of this approach is Sketching [245], which uses a model-checker to generate counterexample traces that are used to refine the space of candidate programs. Deductive synthesis, in contrast, refines a specification to derive the program [195], as discussed in the previous chapter.

While our approach is similar to inductive synthesis, technically it lies midway between inductive and deductive synthesis. We use paths instead of concrete traces and thus are able to capture more of the behavior with each explored “example” path. This is better than concrete inductive synthesis, and leads to practical tools as compared to deductive synthesis. At the same time, it can never reach the formal guarantees provided by deductive synthesis approach. An additional difference is that while previous approaches only refine the space either constructively, through positive reinforcing examples, or destructively, through negative counterexamples, we refine using both positive and negative examples.

Synthesis without formal verifiers Sketching (CEGIS) [245, 246] and even proof-theoretic synthesis described in the previous chapter, rely heavily on formal verifiers. Sketching uses formal verifiers to explain why invalid candidates are not correct and uses the counterexample for invalid candidates to refine the space. Proof-theoretic synthesis encodes the synthesis problem as a search for inductive invariants and therefore needs to infer complicated invariants (and requires a formal verifier with support for such reasoning). In contrast, `PINS` uses symbolic execution and therefore does not require reasoning about complicated invariants.

More on Sketching In terms of the solution strategy, our technique differs from Sketching in four other key aspects. First, the SKETCH compiler uses novel domain specific reductions to finitize loops for stencil [245], concurrent [246], or bit-streaming [247] programs, and is engineered to solve the resulting loop-finitized problem. On the other hand, we finitize the solution space using templates but never finitize loops. Second, we refine at the granularity of paths, while sketching refines using concrete executions and since multiple concrete executions may follow a single path, we are able to cover the space of inputs in fewer iterations. Third, we use SMT reasoning over the correctness constraints to generate concise and small SAT instances that can be efficiently solved, as shown by our experiments, while Sketching uses bit-blasting, which generates formulas that may be difficult, as has been seen by other authors [136]. Lastly, the verification process in Sketching can potentially be testing-based, but it would need to be exhaustive to find the counterexample. On the other hand, we only need to find one feasible path when doing directed symbolic execution to refine the search space. These differences point to important complementary strengths that we intend to exploit in a future SKETCH-PINS hybrid tool.

Chapter 6

Engineering Satisfiability-based Program Reasoning/Synthesis Tools

"Truth is what works."
— William James[1]

In this chapter, we describe the architecture and implementation of our tool set VS^3(Verification and Synthesis using SMT Solvers). This tool set includes the tools $\texttt{VS}^3_{\texttt{LIA}}$ and $\texttt{VS}^3_{\texttt{PA}}$ that implement the theory presented in Chapters 2, and 3. We also use these tools for synthesis, as described in Chapters 4 and 5.

6.1 Using off-the-shelf SAT/SMT solvers

Our invariant inference technique over linear arithmetic (Chapter 2) requires a SAT solver for fixed point computation, while over predicate abstraction (Section 3), we additionally use the theory decision procedures of SMT solvers and their built-in SAT solver. Our approaches to synthesis, proof-theoretic synthesis (Chapter 4) and

[1] American Philosopher and Psychologist, leader of the philosophical movement of Pragmatism, 1842-1910.

`PINS` (Chapter 5), reuse the verifiers built in previous chapters and so use both SAT and SMT solvers.

During the development of the work reported in Chapter 2 we benchmarked various solvers. These included Z3's internal SAT solver [86], MiniSAT [99], ZChaff variants [203], Boolector [45], MathSAT [46, 40, 39], and even a variant that we implemented ourself, based on MiniSAT. While some performed better over certain instances, we found that the heuristics engineered within popular solvers, such as Z3 and CVC3, yielded most predictable results and consistently outperformed most solvers. For the most part, we confirmed that for the instances we were generating the efficiency of the solvers correlated to their performance on the SMTCOMP benchmarks [18]. So while it might be useful in extreme cases to engineer the satisfiability instances at the top-level, for the most part it is sufficient to just rely on the solving capabilities of the best performing solver available in public domain.

For SMT solvers, the results in this dissertation are from runs that use Z3 [86]. We are aware of other comparable solvers, namely CVC3 [21, 19] and Yices [97, 224], which we intend to try in future work.

6.2 Tool Architecture

Both $\texttt{VS}^3_{\texttt{LIA}}$ and $\texttt{VS}^3_{\texttt{PA}}$ use Microsoft's Phoenix compiler framework [1] as a front end parser for ANSI-C programs. Our implementation for each is approximately 15K non-blank, non-comment lines of C# code.

The tool architecture is shown in Figure 6.1. We use Phoenix to give us

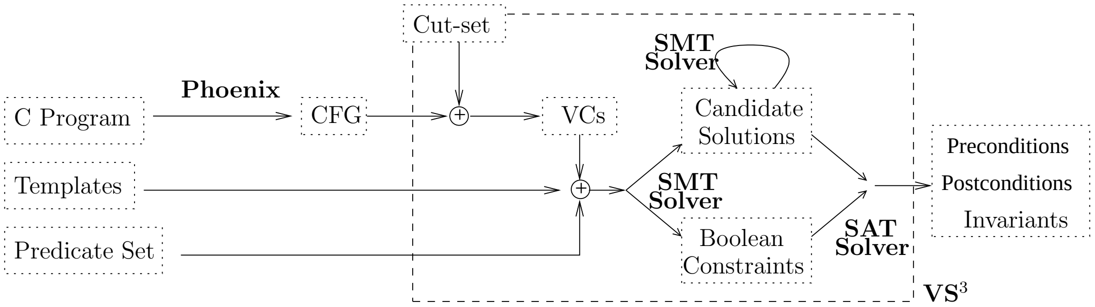


Figure 6.1: The architecture of the $\mathtt{VS}^3_{\mathtt{LIA}}$ and $\mathtt{VS}^3_{\mathtt{PA}}$ tools. In addition to the ANSI-C program (which is replaced with the scaffold when running in synthesis mode), the user provides the templates, the predicate sets, and optionally a cut-set. The user chooses between an iterative and a satisfiability-based fixed-point computation.

the intermediate representation, from which we reconstruct the control flow graph (CFG) of the program. The CFG is then split into simple paths using a cut-set (either generated automatically with a cut-point at each loop header or specified by the user). We then generate a verification condition (VC) corresponding to each simple path. For fixed-point computation the tool provides two alternatives:

- *Iterative fixed-point (Chapter 3)* The iterative scheme performs a variant of a standard dataflow analysis. It maintains a set of candidate solutions, and by using the SMT solver to compute the best transformer it iteratively improves them until a solution is found.

- *Satisfiability-based fixed-point (Chapters 2, and 3)* In the satisfiability-based scheme, a predicate p at location l is identified by a boolean indicator variables $b_{p,l}$. For verification condition vc, we generate the minimal set of constraints over the indicator variables that ensure that vc is satisfiable. These constraints are accumulated and solved using a SAT solver, which yields a fixed-point solution.

For proof-theoretic synthesis (Chapter 4) instead of taking a program as input, the tool takes a scaffold, and instead of using Phoenix to generate the CFG if generates a *template CFG* that is used by the rest of the system. For PINS (Chapter 5), the core tool is just used to find candidate solutions that are valid for all the constraints generated over some paths. The actual PINS algorithm that iteratively refines the space is implemented as a wrapper around the core solver.

6.2.1 Tool Interface

In automatic *cutpoint* mode, VS^3 searches for inductive program invariants at loop headers. Alternatively, in some cases the invariants are simpler if inferred at specific locations, which should form a valid cut-set such that each cycle in the CFG contains at least one location. VS^3 also supports a *manual* mode for user-specified cut-sets.

The user also specifies the *global* invariant template and *global* predicate set for predicate abstraction, as shown. The template is used for invariants at each cut-point, and the predicate set specifies the candidate predicates for the unknowns in the template. We specify the template and predicate set globally to reduce the annotation burden. Specifying them separately for individual cut-points could potentially be more efficient but would add significant overhead for the programmer. We typically we used a predicate set consisting of inequality relations between relevant program and bound variables, and if required, refined it iteratively after failed attempts. In our experience, over the difficult benchmark programs described in previous chapters, coming up with the templates and predicate set is typically easy for the programmer.

6.2.2 Solver Interface

SMT solvers typically provide an API interface that calls the solver to directly manipulate the stack of asserted facts (directly pushing and poping assertions). We currently use the API exported by Z3. We also provide an alternative mode in which all queries are sent to the solver through the SMT-LIB interface, which is a format supported by all major solvers [20].

While Z3 is fairly robust at handling most of the queries we generate, but it has specific limitations that we had to alleviate through mechanisms at the analysis stage before passing the query to Z3. We describe these limitations and our workarounds next. Other solvers have similar limitations.

6.2.2.1 Compensating for limitations of SMT solvers

The generic primitives provided by SMT solvers are expressive but are lacking in some aspects that are needed for our application. We augment the solver by providing a wrapper interface that preprocesses the SMT queries and adds hints for the solver.

Patterns for quantifier instantiation. The current state-of-art for reasoning over quantified facts uses the now commonly known technique of E-matching for quantifier instantiation [85]. E-matching requires *patterns* to match against ground terms.

Because individual SMT queries in our system are over simple quantified terms, a simple heuristic to automatically generate patterns suffices. Given a quantified fact with bound variables $\bar{k}$ and bound boolean term F, we recursively parse F and return valid patterns from F's subterms. A subterm f is a valid pattern if it contains all the bound variables and at least one subterm of f does not contain all the variables. For example, for the fact $\forall k : k > 10 \Rightarrow A[k] < A[k+1]$, we compute the set of patterns $\{\{k > 10\}, \{A[k]\}, \{A[k+1]\}\}$, and for $\forall k : k \geq 0 \wedge k < v \Rightarrow A[k] < min$ we compute the set $\{\{k \geq 0\}, \{k < v\}, \{A[k] < min\}\}$. This simple heuristic is potentially expensive, but allows for automatic and, in practice, fast proofs or disproofs of the implications we generate.

Saturating inductive facts. SMT solvers have difficulty instantiating relevant facts from inductive assumptions. For instance, in our experiments, we encountered assumptions of the form $k_n \geq k_0 \wedge \forall k : k \geq k_0 \Rightarrow A[k] \leq A[k+1]$, from which $A[k_0] \leq A[k_n + 1]$ was needed for the proof. Z3 times out without finding a proof or disproof of whether $A[k_0] \leq A[k_n + 1]$ follows from this assumption. Notice that the pattern $k \geq k_0$ will only allow the prover to instantiate $A[k_n] \leq A[k_n + 1]$ from the ground fact, which does not suffice to prove $A[k_0] \leq A[k_n + 1]$.

We therefore syntactically isolate inductive facts and saturate them. We pattern match quantified assumptions such as the above (consisting of a base case in the antecedent and the induction step in the consequent of the implication) and assert the quantified inductive result. For example, for the case above, the saturated fact consists of $\forall k_2, k_1 : k_2 \geq k_1 \geq k_0 \Rightarrow A[k_1] \leq A[k_2 + 1]$. This, along with the ground term $k_n \geq k_0$, provides the proof.

Theoretically, our approach for saturating inductive facts here is similar to the proposals for axiomatizing reachability using axioms [172, 151, 178, 53]. All these approaches are efficient in practice, but necessarily incomplete, as it is well-known that complete first order axiomatization of transitive closure is impossible [186]. Also related are proposals for simulating transitive closure in first order logic [185]. More details on these approaches can be found in the related work section of a recent paper by Bjørner and Hendrix [32]. In the paper, Bjørner and Hendrix isolate a decidable fragments of a logic that can encode certain forms of transitive closure (appropriate for linked structures, such as lists, and trees) by integrating an LTL checker with an SMT solver. The corresponding combination is a promising direction for future handling of heap structures in our framework.

Explicit Skolemization for $\forall\exists$. Z3 v1.0 does not correctly instantiate global skolemization functions for existentials under a quantifier, and so we must infer these functions from the program[2]. An approach that suffices for all our benchmark examples is to rename the skolemization functions at the endpoints of a verification condition and to insert axioms (inferred automatically) relating the two functions. VS^3 can infer appropriate skolemization functions for the two cases of the verification condition containing array updates and assumptions equating array values. Suppose in the quantified formulae at the beginning and end of a simple path, the skolemization

[2] We are aware of work being pursued in the solving community that will eliminate this restriction. Therefore in the future we will not need to infer skolemization functions.

functions are $\texttt{skl}$ and $\texttt{skl}'$, respectively. For the case of array updates, suppose that locations $\{l_1, l_2, \ldots, l_n\}$ are overwritten with values from locations $\{r_1, r_2, \ldots, r_n\}$. Then we introduce two axioms. The first axiom states that the skolemization remains unchanged for locations that are not modified (Eq. 6.1), and the second axiom defines the (local) changes to the skolemization function for the array locations that are modified (Eq. 6.2):

$$\forall y : (\wedge_i(\texttt{skl}(y) \neq r_i \wedge \texttt{skl}(y) \neq l_i)) \quad \Rightarrow \quad \texttt{skl}'(y) = \texttt{skl}(y) \tag{6.1}$$

$$\bigwedge\nolimits_i \forall y : \texttt{skl}(y) = l_i \quad \Rightarrow \quad \texttt{skl}'(y) = r_i \tag{6.2}$$

For the case of assumptions equating array values, we assert the corresponding facts on $\texttt{skl}'$, e.g., if $\texttt{Assume}(A[i] = B[j])$ occurs and $\texttt{skl}'$ indexes the array B then we add the axiom $\texttt{skl}'(i) = j$.

6.2.2.2 Axiomatic support for additional theories

Modeling quadratics For most of this dissertation we have restricted our constraints to be linear (with propositional connectives) but at times quadratic constraints are critically required. Such is the case for some programs we synthesize (and verify) in Chapter 4, such as a program that computes the integral square root and Bresenham's line drawing algorithm. In this case we provide an incomplete support for handling quadratic expressions.

Our approach consists of allowing the system to contain quadratic expressions, and manipulating them appropriately, e.g., by applying distributing multiplication over addition where required, until the very end when the constraints are required to be solved. At that stage, we provide a sound but incomplete translation of the quadratic constraints to linear constraints.

We rename each quadratic term $a * b$ into a new variable a_b in the constraints to get a linear system from a quadratic system. This modeling is sound because if a solution exists in the new system, it only uses the axiom of equality between quadratic terms. It is incomplete because a quadratic system may have a solution, e.g., using the axiom $a = b \Rightarrow a * a = b * b$, but the corresponding linear system with the renaming, may not have a satisfying solution.

We have found that this sound but incomplete modeling suffices for our programs for the most part. In cases where it does not, we add appropriate assumptions, e.g., `assume`$(a = b \Rightarrow a\_a = b\_b)$, to get consistent solutions on top of the incomplete modeling.

Reachability Some program verification tasks require support for non-standard expressions, e.g., reachability in linked-list or tree data structures. SMT solvers, and in particular Z3, support the addition of axioms to support these kind of predicates.

There are two extra steps in the verification of such programs. First, we define the semantics of field accesses and updates on record datatypes using `sel` and `upd`. A field access $s \rightarrow f$ is encoded as `sel`(f, s), and an update $s \rightarrow f := e$ is encoded as `upd`(f, s, e). Second, by asserting axioms in the solver, we define the semantics of higher level predicates, such as reachability, in terms of the constructs that appear in the program. Let $x \rightsquigarrow y$ denote that y can be reached by following pointers starting at x. Then for the case of reasoning about singly linked lists connected through `next` fields, we augment the SMT solver with the following reachability axioms:

$$\begin{array}{rcll}
\forall x & . & x \rightsquigarrow x & \text{Reflexivity} \\
\forall x, y, z & . & x \rightsquigarrow y \wedge y \rightsquigarrow z \;\;\Rightarrow\;\; x \rightsquigarrow z & \text{Transitivity} \\
\forall x & . & x \neq \bot \;\;\Rightarrow\;\; x \rightsquigarrow (x \rightarrow \texttt{next}) & \text{Step: Head} \\
\forall x, y & . & x \rightsquigarrow y \;\;\Rightarrow\;\; x = y \vee (x \rightarrow \texttt{next}) \rightsquigarrow y & \text{Step: Tail} \\
\forall x & . & \bot \rightsquigarrow x \;\;\Rightarrow\;\; x = \bot & \text{End}
\end{array}$$

For example, using these axioms the solver can prove that $head \rightsquigarrow tail \wedge tail \rightsquigarrow n \wedge n \neq \bot \Rightarrow head \rightsquigarrow (n \rightarrow \texttt{next})$.

6.3 Concurrent reduction

Our algorithms exhibit an embarrassingly parallel structure. For the case of the iterative technique, each individual candidate can be improved in parallel, and for the case of a satisfiability-based technique each verification condition can be reduced to its boolean constraint in parallel. Therefore, we developed a multithreaded implementation of each algorithm. Multithreading is especially natural and useful for the bi-directional satisfiability-based fixed-point computation, which is not restricted to analyzing verification conditions in any particular order.

6.3.1 Super-linear speedup

Our multithreaded implementation achieves super-linear speedup, because it is able to reduce the amount of information computed, using a novel technique which we call *partial solution computation*. This approach generates an *equi-satisfiable* formula that has the same solution but is significantly smaller. By being equi-satisfiable it ensures that the invariant/program solutions computed are identical to what would be computed using the larger formula. To illustrate the redundancy, consider the case of program verification, where an invariant is constrained in similar ways by multiple verification conditions (that start or end at that invariant). A reduction to boolean constraints that is oblivious of this fact computes a significantly larger formula than one that discovers and eliminates redundant clauses. To discover redundancy, we use the notion of partial solution computation.

Partial solution computation The satisfiability-based technique needs to reduce verification conditions into a boolean formula that captures the semantic content of the verification condition. Our multithreaded implementation interleaves these reductions for different verification conditions.

We build on the insight that we can compute *partial solutions* for subformulae, for the case of the final boolean SAT formula being satisfiable, and infer unsatisfiability otherwise. Recall that for the satisfiability-based encoding of VCs in Chapter 3 (Eq. 3.7), we are incrementally computing a SAT instance that is typically small in overall size, but the computation of each individual clause (the second term of Eq. 3.7) involves queries to the SMT solver, and is therefore expensive. We eliminate redundant clauses by using information computed by other threads (working on different reductions) about which indicator boolean variables have been *decided* to be either $true$ or $false$ based on the sub-formula computed so far.

We compute partial solutions for a boolean formula F by checking, for individual boolean variables $b \in \texttt{vars}(F)$, if the formula assigns a truth value to b. We do this by separately checking the satisfiability of $F \Rightarrow b$ and $F \Rightarrow \neg b$. Both of these implications will hold iff the final formula (whose clauses are a superset of the clauses in F) is unsatisfiable. If we find this, we can terminate right away. If, on the other hand, both implications do not hold then we have a consistent sub-formula for which we compute the variables whose values have been *decided*. We remove from consideration all those variables for which neither implication is satisfiable. The partial solution is then the assignment of truth values to the remaining variables as indicated by the satisfiability of $F \Rightarrow b$ ($true$) or $F \Rightarrow \neg b$ ($false$). The correctness of this optimization is due to the following theorem.

Theorem 6.1 (Partial Solution Computation) *Let ϕ be a boolean formula and let $\phi_{\subseteq}$ be a subset of the clauses from ϕ. Then:*

(a) If $\phi_{\subseteq}$ is unsatisfiable then ϕ is unsatisfiable.

(b) If $\phi_{\subseteq} \Rightarrow b$ then any satisfying assignment to ϕ assigns true to b. Correspondingly, if $\phi_{\subseteq} \Rightarrow \neg b$ then any satisfying assignment to ϕ assigns $false$ to b.

(c) If $\phi_{\subseteq} \Rightarrow b \wedge \phi_{\subseteq} \Rightarrow \neg b$ for any b that appears in $\phi_{\subseteq}$, then $\phi_{\subseteq}$ is unsatisfiable.

PROOF:

(a) If $\phi_{\subseteq}$ is unsatisfiable, then no assignment to a superset of the clauses, i.e., ϕ, can assign satisfying values to the clauses that make up $\phi_{\subseteq}$.

(b) Suppose otherwise, i.e., let $\phi_{\subseteq} \Rightarrow b$ and let some satisfying assignment to ϕ assigns $false$ to b. Since $\phi_{\subseteq} \Rightarrow b$ (implicitly quantified over all variables in the formula) holds, and in particular holds for $b \doteq false$, it implies that

$\phi_\subseteq \doteq false$ for all other assignments to the remaining variables. That is $\phi_\subseteq$ is unsatisfiable with b assigned $false$ if $\phi_\subseteq \Rightarrow b$. By Part (a) we know that ϕ is unsatisfiable—contradiction. Therefore, b has to be assigned *true* if $\phi_\subseteq \Rightarrow b$.

A similar argument shows that b has to be assigned $false$ if $\phi_\subseteq \Rightarrow \neg b$.

(c) First observe that Part (b) applies to the degenerate case of ϕ being $\phi_\subseteq$ as $\phi_\subseteq \subseteq \phi_\subseteq$. Now if $\phi_\subseteq \Rightarrow b \wedge \phi_\subseteq \Rightarrow \neg b$ then by Part (b), we know that any satisfying assignment to $\phi_\subseteq$ will assign *true* to b (by the first implication) and it will assign $false$ to b (by the second implication). Both statements cannot be valid together, and consequently we have a contradiction. Therefore, it must be the case that $\phi_\subseteq$ is unsatisfiable.

□

The partial solution computation significantly speeds up the reduction process, when the different threads working on different verification conditions propagate their reductions. The *true* or $false$ assignments for variables whose values have been decided are directly substituted, which typically results in part of the formula being simplified.

Computing maximal solutions using partial solutions The partial solution to the final SAT instance can be used to compute the greatest or the least fixed-point solution. For the boolean variables that are not in the partial solution any truth assignment corresponds to a valid invariant. Therefore, by assigning $false$ to the variables of a negative unknown and *true* to the variables of a positive unknown we get a least fixed-point. The opposite assignment yields a greatest fixed-point. In practice, we do not care about the optimality of the solution generated by the satisfiability-based approach and therefore have not implemented this last greatest/least-fixed point optimization.

6.4 Summary

Building on the theory described in Chapters 2—5, in this chapter we described the implementation challenges of building a tool for program reasoning and program synthesis. The tool can infer expressive properties of programs using minimal annotations in the form of invariant templates, and can also synthesize programs with minimal descriptions, given by the user.

Chapter 7

Extensions and Future Work

"Heavier-than-air flying machines are impossible."

— Lord Kelvin[1]

This dissertation focuses on program reasoning and program synthesis for the case of sequential, imperative programs. There are three sets of extensions that we plan to address in the future. The first set consists of augmenting the *expressiveness* of our schemes for reasoning and synthesis while still remaining in the domain of sequential, imperative programs. The second set consists of applying and developing techniques for reasoning about and synthesizing programs and proofs in *non-(sequential, imperative)* domains. The third set consists of treating synthesis as *augmenting compilation*, where we attempt to synthesize modules that plug into legacy code such that the new program meets desired specifications.

[1] President of the Royal Society, 1895.

7.1 Expressiveness

Linear Arithmetic The work described in Chapter 2 can be extended in at least two directions. The first one is to extend these techniques to discover a richer class of invariants involving arrays, pointers, and even quantifiers. The technical details of these extensions have already been worked out, and we are currently in the process of implementing these ideas in our tool. Second, we are investigating use of new constraint solving techniques, in particular QBF (Quantified Boolean Formula) solvers. This would alleviate the need for applying Farkas' lemma to compile away universal quantification, leading to smaller sized SAT formulas, but with alternating quantification. While in general QBF is PSPACE-complete, and therefore we would expect these instances to be fairly difficult to solve, it may be that for limited classes of instances the QBF formulae are efficiently solvable, similarly to the use of SAT/SMT solvers in this dissertation.

Predicate Abstraction In Chapter 3 we restricted ourselves to simple theories supported by SMT solvers. In particular, we most extensively use the theory of arrays (that too without extensionality, which states that $\forall A, B : \forall i : (A[i] = B[i] \Rightarrow A = B)$ [44, 250]), uninterpreted functions, and linear arithmetic, which are all basic theories supported by all solvers. Today, SMT solvers in fact support many more theories efficiently. For instance, we added incomplete support for reachability and were able to verify small linked-list programs. There have been recent proposals, that incorporate a logical theory for unbounded reachability within an SMT solver, which can potentially be used directly to verify *heap manipulating* programs [224].

In particular, we intend to try verifying the full functional correctness of list/tree and other data structure operations (e.g. insertion in AVL/Red-Black trees) within our satisfiability-based framework for reasoning. Additionally, such extensions will also allow the synthesis of heap manipulating programs.

Another important consider is that of *abstraction refinement* [59]. Ideas from counterexample guided refinement can be incorporated in our framework to build a system that supports automatic predicate discovery. More interestingly, instead of traditional iterative approaches to predicate inference (e.g., the maximal solution computation in Chapter 2), it should be feasible to encode the synthesis of predicates as solutions to a satisfiability instance.

Modular Synthesis Program synthesis as we have considered synthesizes the entire program corresponding to a given functional specification (Chapter 4) or related program (Chapter 5). In fact, even previous approaches take a similar end-to-end approach to synthesis [245]. However, the eventual success of automated synthesis will lie in its ability to synthesis programs in terms of an abstract interface corresponding to lower level functions.

We implicitly explored this issue through the use of predicates over uninterpreted functions (with externally defined semantics) in proof-theoretic synthesis. An instance of this was the use of definitional functions (and axioms) for the case of dynamic programming programs; or the use of uninterpreted functions modeling the layout of two dimensional arrays; or the use of the `swap` predicate for sorting programs. While these demonstrate the feasibility of synthesis over an abstract interface to lower level functions, they are not modular synthesis. In particular, a defining feature of modular synthesis is the ability of the system to automatically infer what functions implement which functionality, i.e., the interface boundary. For the discussion in this dissertation, we had the interface boundary manually specified by the user. Inferring the interface boundary is a key technical challenge that needs to be addressed.

7.2 Applications to non-(sequential, imperative) models

Cross-synthesis: Architecture-specific synthesis In Chapter 4 we proposed the use of resource constraints to restrict the space of candidate programs. We envision using resource constraints to focus attention to certain classes of computations, instruction sets, and memory access patterns, such as those allowed by peculiar architectures, e.g., Cell Broadband Architecture [104], GPUs [213] for which the CUDA [209] and OpenCL [248] programming models have been proposed. We would define the synthesis problem as taking a program in an standard unrestricted programming model, and the synthesizer would generate the corresponding semantically equivalent program in the restricted programming model.

Concurrency Recent work on local reasoning for concurrency [256, 94, 106] has the flavor of interprocedural summary computation, but instead of computing summaries for procedures computes summaries of interference behavior of threads. Our goal-oriented satisfiability-based invariant inference approach is particularly suitable for interprocedural summary computation and therefore has potential to be useful for thread interference summary computation as well. Using precise interference summaries, thread modular reasoning can facilitate verification and synthesis of concurrent programs.

Synthesizing functional programs A inference technique for dependent types can be used to synthesize functional programs in the same way program verifiers can synthesize imperative programs. For it to be useful for synthesis, inference is necessarily required to be annotation-less as annotations tag given programs. Proposals for limited-annotation limited dependent-type inference [229, 161, 252] have the potential to be used for synthesis of functional programs.

Additionally, Appel described how Single Static Assignment (SSA [230, 4]) style is essentially functional programming [6], and we know that continuation passing style (CPS)—the intermediate representation of choice for functional program compilers—and SSA are formally equivalent, and optimizations formulated for one are directly applicable to the other [162]. Hence it may be possible to use the techniques we developed here directly for synthesizing functional programs by suitable representational translation.

Synthesizing proofs of progress and preservation A more radical application of synthesis could be to the domain of "proof-synthesis." When designing a type-system, the method of choice for proving its correctness is to use an operational semantics approach and prove progress and preservation [214]. The key difficulty in such proofs is the inference of a suitable induction hypothesis. With a correct hypothesis the proof typically is mostly mechanical with case splits based on the structure of the language. We can pose the problem of induction hypothesis inference as invariant inference and the proof cases as imperative paths that need to be synthesized.

7.3 Synthesis as augmenting compilation

Program reasoning and synthesis may be defined over a fragment of the total program. We comment on such possibilities here.

Synthesizing correctness wrappers We propose synthesizing only fragments of code that serve as wrappers around otherwise potentially incorrect programs. Given a specification of correctness (lack of crashes, no information leaks, etc.), and a program that potentially does not meet the specification, the task would be to synthesize a wrapper that calls into the raw programs and modifies its behavior at appropriate locations such that it meets the specification.

One application may be to information flow security. Consider a browser that can potentially leak information through Javascript. For every location in the browser source code where a call is made into the Javascript engine, we synthesize and insert a sanitization function that ensures that only low security data passes through. Another application may be in making distributed computation robust. In this case, the wrapper would serve as a monitoring state machine that terminates, starts, or restarts computation on detecting anomalous behavior. Another potential application is to proof-carrying code (PCC) [207]. In traditional PCC, the client has a specification and the developer is responsible for sending a certificate along with the program, and the client verifies the certificate to check if it meets the security policy. We can imagine the client sending a (sanitized) version of his policy to the developer such that the developer only writes a partial program, and the synthesizer fills out the remainder such that the resulting program is guaranteed to meet the specification.

Synthesizing aspects (cross-cutting concerns) Aspect-oriented programming [163] defines a programming model in which the program's functionality is divided not by lexical boundaries but by semantic similarities of various fragments. For instance, authorization and logging are typical cross-cutting concern [109]. While aspects can lead to cleaner software if used well, they can also leads to fragmentation of code away from the data, e.g., code manipulating a variable could be in multiple aspects that are scattered all throughout the codebase, possibly far away from the class owning the variable. We can imagine a programming model in which the only allowable aspects are the ones that the synthesizer generates. In such a scenario the only codebase available to the developer is the one that is localized, removing any maintainability concerns of aspects. The aspects would be suitably synthesized (and be correct) for any change to the codebase made by the developer as the code corresponding to the aspect will never be directly modified.

Synthesizing "failsafe"s Programs are rarely reliable or robust. While we can verify their correctness, or lack thereof, using the reasoning techniques developed in this dissertation, we can potentially also synthesize bypass mechanisms that ensure that failing programs are sandboxed. Similar to failure-oblivious computing [228], but more semantically aware, such a wrapper would keep track of out-of-bounds reads and writes and instead of indiscriminately allowing them, would consider the changes in program behavior from a given baseline and suitably change values to match statistically more probable program states.

Synthesizing attackers An interesting application to security verification may be to model the attacker as an unknown state machine (potentially with an unbounded state space). Then using the techniques described in this thesis, we can imagine defining a specification of a bad state, i.e., defining the existence of an attack. We then synthesize an attacker such that its combination with the program under consideration meets the specification, i.e., shows the existence of an attacker and corresponding attack.

Chapter 8

Related Work

> *"The history of mankind is the history of ideas."*
>
> — Luigi Pirandello[1]

The work in this dissertation builds on significant advances in programming languages theory in the last few decades. We review a tiny fraction of that related literature in this chapter.

8.1 Program Reasoning

The desire to do precise program reasoning is not new. Foundational and widely accepted frameworks in which program analyses can be formulated include Kildall's data-flow analysis [164], Cousot and Cousot's abstract interpretation [72], and Clarke, Emerson, and Sifakis' model checking [98]—all of which perform iterate approximations of program properties. This dissertation builds on a relatively more

[1] Italian short-story Playwright, Writer, Dramatist and Novelist, who was awarded the Nobel Prize in Literature in 1934 for his "bold and brilliant renovation of the drama and the stage," 1867-1936.

recent non-iterative constraint-based framework proposed by Manna et. al. [62, 235]. A constraint-based framework allows building analyses that assume templates to encode program semantics as finite constraints.

The history of program reasoning—verification and property inference—is vast and varied, and we will necessarily be unable to cover all related work. We discuss the ones most relevant to our work in this dissertation.

8.1.1 Program Verification

For program verification, we consider a somewhat linear progression based on the technical difficulty of techniques based on invariants.

8.1.1.1 Invariant validation using SMT solvers

The first towards formally verified software does not even talk of invariant *inference*. Even without inference, the task of just *validating* user-provided invariants is non-trivial. The difficulty in invariant validation comes from discharging complicated invariants, which could be quantified, making the verification condition discharging process undecidable in general. Before the advent of SMT solvers, either custom theorem provers were used, or domain-specific decision procedures for limited forms of invariants were used.

With the increase in size of software, resulting in a more significant need for formally correct components, invariant modeling languages have gained popularity. Microsoft's Spec# [17] and Dafny [182], the Java Modeling Language

(JML) [50], ESC/Java [111] are examples of such languages. Similar user-provided invariant checking approaches exist that validate very expressive program properties by exploiting the power of SMT solvers. In particular, they leverage the ability of these solvers to reason about formulae over combinations of different theories. These approaches essentially treat SMT solvers as limited forms of theorem provers. Approaches in this domain include checkers for loop optimization [150], arbitrary C assertions [240], low-level systems code [64], and even concurrency properties [179]. There are also larger frameworks in which analyses can be written, e.g., the Why/Krakatoa/Caduceus deductive verification system [108], or Boogie/PL [183, 16].

Verifiers of this form work with the assumption that an external oracle exists that generates the difficult parts, i.e., invariants, in the proof required for verification. This external oracle could be a human programmer or a proof-generating compilation step. The system then generates constraints over the invariants using the program, and the SMT solver is used to discharge these constraints, validating the externally provided invariants. These projects address a question that is complementary to this dissertation. We talk of invariant and program inference, while these validation approaches use the result of inference (from techniques such as ours) and verify much larger codebases.

8.1.1.2 Invariant Inference over Linear Arithmetic

While invariant validation techniques are directed towards scalability, invariant inference targets expressivity. The guiding objective for invariant inference technology is the dream of fully automatic full functional verification. So while it may be possible with invariant validation to formally prove a particular piece of software correct, when moving to the next piece of software, we have to start from scratch. On the other hand, if we succeed in building automatic inference techniques for expressive invariants, then each successive piece of software does not require proportional human effort. Therefore the benchmarks in this field consist of small but complicated programs that require inference techniques for very expressive invariants. The hope is that if the techniques work for these programs, then for larger programs the reasoning required will still be within the reach of the tool. Linear arithmetic is one tractable, yet expressive domain for which inference techniques have been designed.

Techniques based on abstract interpretation Cousot's abstract interpretation is a foundational framework for specifying program property inference as iterative approximations over a suitable domain (a lattice of facts in which the invariants are expected to lie) [72]. Using abstract interpretation, sophisticated widening techniques [125, 126], abstraction refinement [264, 132], and specialized extensions (using acceleration [124], trace partitioning, and loop unrolling [34]) have been proposed for discovering conjunctive linear inequality invariants in an intraprocedural setting. Leino and Logozzo also propose introducing a widening step inside SMT solvers to generate loop invariants [184]. For disjunctive domains, powerset extensions over linear inequalities have been proposed [118, 133]. There are also alternative approaches that exploit the structural correlations between the disjunctive invariant and the control flow structure for disjunctive invariant inference [232, 30]. All these are specialized to work for specific classes of programs. In contrast, the satisfiability-based approach we propose in Chapter 2 can uniformly discover precise invariants in all such classes of programs with arbitrary boolean structure, if required. While an iterative approach can be advantageous for weakest precondition and strongest postcondition inference, where we desire to compute the extremum of the sub-lattice making up the fixed-points, for the case of verification where any fixed-point suffices, a satisfiability-based approach offers significant advantages: It is goal-oriented and thus does not compute facts that are redundant to the assertions being proved.

In the interprocedural setting, there has been work on discovering linear equality relationships for interprocedural verification [231, 204]; however the problem of discovering linear *in*equalities is considered difficult. Very recently, some heuristics for linear equality relationships have been proposed by extending earlier work on transition matrices and postponing conditional evaluation [238]. The precision of these techniques is unclear in the presence of conditionals. The approach in Chapter 2 handles disjunctive reasoning seamlessly, and it can discover linear inequalities interprocedurally as precisely as it can intraprocedurally. The approach is goal-oriented and so the system only discovers relevant summaries that are required for verification of call sites. Additionally, abstract interpretation based summary computation needs to iterate multiple times to ensure the summary is as weak in the pre- and as strong in the postcondition as required. We have not experimented with interprocedural benchmarks over predicate abstraction, but we believe the satisfiability-based technique should possess the same theoretical benefits as over linear arithmetic.

Techniques based on constraint solving Theoretical expositions of program analysis techniques frequently formulate them as constraints (constraint-based CFA [210], type inference [221], reachable states in abstract interpretation [72], and model checking [98] among others) and typically solve them using fixed-point computation. We are not concerned with techniques such as those here, but instead with techniques that use a constraint solver at the core of the analysis, i.e., those that reduce the analysis problem to constraints to be solved by either mathematical, SAT, or SMT solvers. Constraint-based techniques using mathematical solvers, have been successfully used to discover *conjunctive* linear arithmetic invariants by Manna et. al. [62, 234, 233, 235] and by Cousot [75]. The satisfiability-based approach presented here can be seen as an extension of these constraint-based techniques and can handle invariants with arbitrary, but pre-specified, boolean structure and also in a context-sensitive interprocedural setting—partly because we use a SAT solver at the core instead of mathematical linear programming solvers.

Constraint-based techniques have also been extended for discovering non-linear polynomial invariants [160] and invariants in the combined theory of linear arithmetic and uninterpreted functions [28], but again in a conjunctive and intraprocedural setting. It is possible to combine these techniques with our formulation to lift them to disjunctive and context-sensitive interprocedural settings.

Constraint-based techniques, being goal-directed, work naturally in program verification mode where the task is to discover inductive loop invariants for the verification of assertions. Otherwise, there is no guarantee on the precision of the generated invariants. Simple iterative strategies of rerunning the solver with the additional constraint that the new solution should be stronger, as proposed by Bradley and Manna [41], can have extremely slow progress, as we discovered in our experiments. Our approach for strongest postcondition provides a more efficient solution. Additionally, we present a methodology for generating weakest preconditions.

Other approaches can also be viewed as being constraint-based, e.g., SATURN [269], which unrolls program loops a bounded number of times, essentially reducing the program analysis problem to a circuit analysis problem that has a direct translation to SAT. SATURN has been successfully used for bug finding in large programs [93]. In contrast, the approach in Chapter 2 can potentially find the most-general counterexample and can also find bugs in programs that require an unbounded or a large number of loop iterations for the bug to manifest.

Proofs and counterexamples to termination Termination analysis is an important problem with the potential for significant practical impact. The primary approach to proving termination properties in imperative programs is through ranking functions for each loop. Ranking functions impose a well-founded relation on the iterations of a loop, proving its termination. Work by Colon and Sipma [63], Podelski and Rybalchenko [219], Bradley et. al. [42, 43], Cousot [75], and Balaban et. al. [9] made key strides in inferring linear ranking functions. The Terminator project incorporates many of these ideas and others into a usable system for proving termination of systems code [24]. The SPEED project attempts to tackle a harder problem, that of computing symbolic bounds for loops and recursive functions [138]. Such an analysis can be used to bound resource usage, including time, space, and communication. On the flip side, techniques can attempt to find counterexamples to termination, i.e., evidence of non-termination, such as the approach by Gupta et. al. [142]. Their technique finds counterexamples to termination properties by identifying *lassos* (linear program paths that end in a non-terminating cycle) and using a constraint solving approach to find recurring sets of states.

The approach for bounds analysis in Chapter 2 is one solving technology that can be applied towards bounds, termination and non-termination analysis. Additionally, by inferring maximally weak preconditions, the approach can also be used for conditional termination analysis, where we infer preconditions under which the program terminates. Our scheme for proving non-termination is more direct than previous proposals and can potentially find the most-general counterexample to termination.

8.1.1.3 Invariant Inference over Predicate Abstraction

Template-based analyses The template-based approach used in this work is motivated by recent work on using templates to discover precise program properties, such as numerical invariants by Manna et. al. using mathematical solvers [233, 234, 62], Kapur using quantifier elimination [160], Beyer et. al. for the combination with uninterpreted functions [28], Gulwani et. al.'s use of templates for quantified invariants in an abstract interpretation framework [137]. All these techniques differ in expressivity of the templates, as well as the algorithm and underlying technology used to solve for the unknowns in the templates.

Except for Gulwani et. al.'s work, all the other techniques employ a constraint-based approach to encode fixed point, reducing invariant generation to the task of solving a constraint. However, these techniques use specialized non-linear solvers. On the other hand, we use SAT/SMT as our core solving mechanism. We perceive that mathematical solvers are an overkill for the discrete constraint solving task at hand. Gulwani et. al. use an iterative least-fixed point approach; however, it requires novel but complicated under-approximation techniques.

Predicate abstraction Predicate abstraction was introduced in the seminal paper by Graf and Saidi showing how quantifier-free invariants can be inferred over a given set of predicates [128]. Since then the model checking community, e.g., in the SLAM model checker [15], in the MAGIC checker [2], and Das and Dill's work [82, 80], made significant strides in the use of predicate abstraction as a very successful means of verifying properties of infinite state systems.

Our templates in Chapter 3 range over conjunctions of predicates wrapped in an arbitrary boolean structure. This is in contrast to the integer coefficients we discover in Chapter 2 for a linear arithmetic template. Our predicate abstraction template is inspired by important work on predicate abstraction in the model checking community [112]. Efforts to improve the expressivity of predicates used by these systems included Lahiri's indexed predicates, which contain free variables that are implicitly quantified and so can express limited sets of quantified properties [176]. Podelski and Wies applied the idea of indexing to predicates over the heap to reason about heap manipulating programs in the context of predicate abstraction [220]. Our work extends those ideas to include an arbitrarily expressive, explicitly indexed, boolean structure over the predicates. Additionally, since our transfer functions are direction-agnostic, and in particular not necessarily forward, we can define weakest precondition analyses as well, which is not straightforward for previous abstract interpretation-based definitions of the forward transfer functions.

In this dissertation, we have not considered the orthogonal problem of computing a set of predicates that is precise enough to prove the desired property. Automatic abstraction refinement, i.e., predicate discovery, has been critical in making predicate abstraction based model checking mainstream. Counterexample guided abstraction refinement (CEGAR) by Clarke et. al. is one core iterative approach that facilitates predicate discovery [59, 57]. In CEGAR, the model checker attempts verification using the given abstraction, and if it fails a counterexample is produced that helps infer predicates that refine the abstraction. Craig interpolation has been applied to the counterexample path to discover appropriate predicates [147, 154]. Improvements to the core interpolant scheme have since been developed [95], and approaches for doing it lazily are known [148]. We currently do not address this issue and instead assume that the set of predicates is provided. As future work, it would be interesting to see how our technique can be combined with predicate discovery techniques.

Computing optimal transformers Our iterative fixed-point algorithms in Chapter 3 can be seen as computing the best transformer in each step of the algorithm. These abstract transformers are over a lattice defined by the predicates and template. For the case of domains other than predicates, Reps, Sagiv, and Yorsh designed decision procedures for such best abstract transformers [227].

Dependent types for assertion checking Types are coarse invariants, as they represent facts that hold of the values stored in the typed variables. Types start resembling specifications and invariants when we introduce the notion of dependent typing [8]. In dependent typing, the types of variables can be qualified by arbitrary expressions. In typical proposals the dependent types are provided by the user (and can possibly be validated by the type-checker) [23, 268, 65], which is similar to the scenario of validating user-provided invariants.

One form of this qualification is using refinement types [116] where the standard ML type, e.g., `int`, is refined by a predicate, e.g. a refinement indicating positive integers may be $\{\nu : \texttt{int} | \nu > 0\}$. For refinement types, which are restricted dependent types, inference proposals exist by Knowles and Flanagan [168], by Rondon, Kawaguchi, and Jhala [229, 161], and by Terauchi [252]. These proposals can be viewed as alternative type-based proposals for invariant inference.

Symbolic model checking McMillan made a fundamental breakthrough in model checking by introducing the notion of symbolic model checking [60, 49]. Symbolic model checking uses ordered BDDs to represent transitions implicitly and without explicitly expanding the state graph [47]. Symbolic model checking is able to explore on order of 10^{20} states. The implicit symbolic representation also means that program states are abstracted and fixed-point iteration is required to infer properties of infinite state systems.

8.1.1.4 Verification without invariant inference

Model checking Traditional model checking [98, 56], i.e., non-symbolic model checking, checks whether a system meets its specification by writing the system as a Kripke structure, i.e., a transition system with property labels on the states, the specification as temporal logic formula, and checking that the Kripke structure is a model of the temporal logic formula. The last step, model checking, is done through explicit state exploration that labels the states with properties. While model checking typically encounters a space explosion problem, various algorithmic techniques have been designed to efficiently explore the space, and significant engineering effort has helped realize practical verification systems using this approach. Notice that the only formal statement required is the specification formula (given in a suitable logic, such as LTL or CTL), and thus potentially any specification that is expressible is checkable. This is not the case when we attempt to infer invariants, which are from limited domains and thus failure to infer invariants indicates either that the domain is not expressive enough or that the program is faulty. While using invariants introduces the possibility of restricting the class of verifiable programs, the benefits significantly outweigh the costs, as was realized by the model checking community with the advent of symbolic model checking, which requires fixed-point computations.

Approximate verification Program testing, be it concrete, symbolic [165], or a combination such as concolic [239, 121], can be viewed as an approximation to formal verification. These techniques do not infer invariants and are necessarily incomplete in the presence of loops. Testing attempts to explore as many paths through the program as possible and ensure that on each path the specification is met. While more practical for software developers that are unwilling to deal with formal specifications, they lack formal guarantees, but have the advantage of being less demanding on theorem proving resources. In fact, our synthesis approach in Chapter 5 inherits both the advantages and disadvantages of an invariant-less technique.

Random interpretation combines ideas from testing with abstract interpretation to yield a technique that may be unsound in addition to being incomplete, but the unsoundness is probabilistically bounded [139, 140, 141]. Random interpretation alleviates the tension of exploring multiple different paths, by combining/joining them using ideas from abstract interpretation. The join is probabilistic (unlike traditional abstract interpreters whose join function is deterministic) and is inspired by ideas from randomized algorithms. Using the novel join functions, random interpretation yields probabilistic sound analyses.

8.1.2 Specification Inference

Strongest postcondition and weakest precondition inference Abstract interpretation works by iteratively generating a better and better approximation to the desired invariants [72]. Theoretically, the core operators on the domains can be defined such that they either compute the strongest or weakest invariants. In practice, strongest postcondition inference is tractable to compute and thus most verification techniques defined using abstract interpretation compute the strongest postcondition and then check if the assertions in the program hold under that postconditions. Weakest precondition inference typically generates too many uninteresting preconditions, making its use troublesome. In our work here, the use of templates restricts attention to preconditions of desired forms.

Chandra, Fink, and Sridharan do propose a scalable heuristic technique for generating useful preconditions in Java programs, but get past the difficulty of handling loops by using user-annotations [52].

Precise summary computation Precise specification inference has the potential to facilitate modular analyses but is relatively unexplored. Yorsh, Yahav, and Chandra propose an approach that combines abstract micro-transformers [271], while Gulwani and Tewari propose an abstract interpretation-based framework for computing symbolic summaries [134]. Yorsh et. al.'s approach is compositional, and Gulwani and Tewari's approach computes weakest preconditions for *generic* (symbolic) assertions and then unifies them. Both show the applicability to specific abstract domains; Yorsh et. al. consider the typestate domain and Gulwani et. al. consider uninterpreted functions and linear arithmetic. Both attempt to compute the most precise summaries for procedures, and this may be too expensive. Our techniques on the other hand, are goal-oriented in that they do global interprocedural analysis and compute only the summaries that are required for the verification of the call sites and additionally works over any domain for which a satisfiability-based analysis is available.

8.2 Program Synthesis

The desire to automatic synthesize programs is also not new, although much less research effort has been directed towards synthesis as compared to program reasoning. While the problem was called a "dream" by Manna and Waldinger in 1979 [190], and defined in the context of model realizability by Pnueli and Rosner in 1989 [218], the worst-case complexity of program synthesis hampered progress. Statements such as "one of the most central problems in the theory of programming" and "programming is among the most demanding of human activities, and is among the last tasks that computers will do well" in the above papers, served both to promote and relegate program synthesis to being an unachievable dreams. It is 2010, and our view of automatic program verification has changed from being intractable to being realizable. Correspondingly, it is time to revise our view of automatic program synthesis from being impossible to being plausible. While we are not claiming program synthesis is theoretically any easier now, the advent of powerful program reasoning techniques gives us hope that this technology can be used for program synthesis—as we do directly in Chapter 4 and indirectly in Chapter 5.

The primary reason for the skepticism towards program synthesis is that an automated tool is unlikely to discover the “intuition” behind solving a problem. Human developers find these insights and encode them in programs that meet a certain specification. What we argue in this dissertation is that automatic program synthesis tools need not discover “intuition” but instead need to find just one solution that meets the specification—one that is formally correct but may not be the elegant solution a human developer may design. This is similar in spirit to program verification, where the human developer may find an insightful proof while an automated tool finds any valid proof that suffices, and this proof may not be elegant or even readable.

In the alternative perspective of *providing* the tool with the insight and having it fill out the details, significant work has been done. Previous approaches can be categorized as either *deductive* or *inductive*. We refer the reader to a recent survey describing the various categorization of synthesis approaches as deductive (constructive), schema-guided, or inductive [22].

8.2.1 Deductive Synthesis

Deductive synthesis is the approach of successively refining a given specification using proof steps, each of which corresponds to a programming construct. By having the human developer guide the proof refinement, the synthesizer is able to extract the insight behind the program from the proof.

Most of the work in deductive synthesis stems from the seminal work of Manna and Waldinger [194, 195]. Successful systems developed based on this approach include Bates and Constable's NuPRL [66] system, and Smith's KIDS [243], Specware [199], and Designware [244] systems. In these systems, the synthesizer is seen as a compiler from a high-level (possibly non-algorithmic) language to an executable (algorithmic) language, guided by the human. To quote Smith, "the whole history of computer science has been toward increasingly high-level languages--machine language, assembler, macros, Fortran, Java and so on—and we are working at the extreme end of that."

While such systems have been successfully applied in practice, they require significant human effort, which is only justified for the case of safety/mission-critical software [100]. As such, these systems can be viewed as programming aids for these difficult software development tasks, somewhat related to the idea of domain-specific synthesizers such as AutoBayes for data-analysis problems [110], StreamIt for signal-processing kernels [253], or Simulink for hardware synthesis [26].

We categorize proof-theoretic synthesis from Chapter 4 as midway between deductive and schema-guided synthesis. Schema-guided synthesis takes a template of the desired computations and generates a program using a deductive approach [114]. Some heuristic techniques for automating schema-guided synthesis have been proposed, but they cater to a very limited schematic of programs, and thus are limited in their applicability [96]. Schema-guided synthesis specialized to the arithmetic domain has been proposed using a constraint-based solving methodology [61]. Our technique in Chapter 4, if viewed as a schema-guided approach, formalizes the requirements for it to work over any domain, as opposed to particular instances, e.g., linear arithmetic as considered previously [61]. Additionally, while the specification of the program synthesis task is comparable to these approaches, the satisfiability-based efficient solving methodology is novel in our approach.

8.2.2 Inductive Synthesis

Inductive synthesis is the approach of generalizing from instances to generate a program that explains all instances or traces that meet a specification. The instances could be *positive* ones that define valid behavior or counterexamples that eliminate invalid behavior.

Of particular note in this category is the work by Bodik and Solar-Lezama et. al. on the Sketch system, which synthesizes from partial programs [245]. Their work has helped revive interest in practical synthesis in recent years, while still having the human programmer provide the insight behind the program in the shape of a "sketch" of the desired computation. The Sketch system fills out integer holes, whose values may be difficult for the programmer, in a partial program and as such is also a programming aid. Bodik, Solar-Lezama et. al. deserve significant credit for designing a synthesis interface that software developers will be comfortable with. The approaches we present in this dissertation derive much inspiration from their work and, in fact, both proof-theoretic synthesis and `PINS` go through intermediate representations that resemble a sketch of the desired program, albeit with holes that are filled in by full expressions rather than just integers.

Combinatorial sketching does not use a mathematical formulation, but instead uses another, unoptimized program as the specification of the desired computation [245, 246, 247]. A model checker eliminates invalid candidate programs—by matching the candidates behavior against the that of the unoptimized program—that the synthesizer enumerates heuristically using a guided search. Loops are handled incompletely, by unrolling or by using a predefined skeleton. Arguably, software developers are more comfortable with partial programs with holes than with formal specifications, and this was the motivating factor behind the design of the Sketch system. While such a design choice makes program synthesis accessible, which is very important, but at the same time, it limits the technical machinery that can be applied to "resolve" the sketch. In particular, the lack of a formal specification of the intended behavior means that proof-theoretic synthesis cannot directly be applied to solving sketches. On the other hand, `PINS` can certainly be used to resolve sketches—possibly more efficiently than using a counterexample generating model checker or even combined with the existing solution strategy.

Recently, a novel approach for synthesis of bit vector programs using input-output examples has been proposed [152]. The techniques assumes the presence of an oracle, e.g., a human user, that is queried by the system for the validity of an input-output pair. The information from the oracle is used to guide the search and prune it appropriately until only a single solution remains. In the context of using traces to prune the search space, this approach is similar to Sketching (that uses concrete counterexample traces), and to a lesser degree to `PINS` (that uses symbolic traces). It is different from Sketching in that it can use both positive and negative instances to prune the search space. It is different from PINS in that it works for acyclic program fragments while PINS automatically decides which traces to explore in a program with loops.

8.2.3 A Liberal View of Synthesis

Deriving programs with proofs Dijkstra [92], Gries [130], and Wirth [267] advocated that programmers write programs that are correct by construction by manually developing the proof of correctness alongside the program. Because techniques for efficient invariant inference were unavailable in the past, synthesis was considered intractable. For instance, Dijkstra wrote, “I should [*sic*] like to stress that by using the verb ‘to derive’ I do not intend to suggest any form of automatism [*sic*], nor to underestimate the amount of mathematical invention involved in all non-trivial programming. (On the contrary!) But I do suggest the constructive approach sketched in this paper as an accompanying justification of his inventions, as a tool to check during the process of invention that he is not led astray, as a reliable and inspiring guide.” [91] While automation was unavailable when Dijkstra wrote this, theoretical and engineering developments since then indicate that synthesizing programs and proofs simultaneously may be possible.

Extracting program from proofs The semantics of program loops is related to mathematical induction. Therefore, an inductive proof of the theorem induced by a program specification can be used to extract a program [195]. Using significant human input, theorems proved interactively in the Coq have a computational analog that can be extracted [25]. The difficulty is that the theorem is of the whole program, and proves that an output exists for the specification. Such a theorem is much more difficult than the simple theorem proving queries generated by the verification tool. Additionally, it is hard to generate *good* code since the notion of a good proof is hard to define.

Model checking-based synthesis of reactive systems Perhaps the most directly related work on fully automatic program synthesis are the proposals from the model checking community for automatic synthesis of reactive systems. See Moshe Vardi's slides for an overview [257]. Here synthesis is interpreted as the realizability of an linear time logic (LTL) specification of the system. While it has been shown that synthesis in this manner is decidable, the complexity is doubly exponential [217]. (One exponent comes from the translation of the specification to a Büchi automata, and the second comes from determinization.) Since these results were discovered, significant effort has been spent on optimizing constructions [156]. For limited classes of systems, e.g., supervisory controller synthesis [225], and controller synthesis to timed systems [7], linear time results were shown. While these results show promise for the case of *circuit* synthesis (the synchronous case), they do not directly translate to programs (the asynchronous case). A reduction from the asynchronous to the synchronous case incurs unacceptable exponential blowup [218]. Recent work in the domain attempts to both over-approximate and at the same time heuristically underapproximate to infer the realizability of the specification.

Hardware synthesis Synthesizing circuits is a theoretically easier, but still very challenging, task compared to program synthesis. Circuit synthesis has also been explored more deeply. First described as Church's problem [55], it has more recently been addressed in the model checking community with mixed success [35, 37]. Practical tools that can synthesize Verilog descriptions from specifications have been built [157, 196]. Due the lack of loops, the hardware synthesis problem does not encounter the hurdles that we had to overcome. The work presented in this dissertation has different technical challenges and so we defer giving a more detailed account of work on hardware synthesis, but refer the reader to discussions elsewhere on Church's problem [254], and on hardware synthesis [170, 78, 242].

Program repair and game-based synthesis Synthesis can be viewed as a game. The idea is to define to a game between the environment and the synthesizer where the winning strategy for the synthesizer corresponds to the synthesized program [158]. Henzinger et. al. have explored *quantitative synthesis*, where instead of asking only whether a program meets the specification, they also ask how close is its behavior to the specification [36]. Such an approach has been applied to the synthesis of robust systems [38], for fault-localization and fixing [155], and to C programs using predicate abstraction [131].

Deriving inverses as domain-specific synthesis Previous strategies for deriving program inverses can be categorized into two classes. The first are strategies that require the complete proof for the original program (conceptually a proof of injectivity), from which they provide proof rules to syntactically construct the inverse [89, 54, 130, 101]. However, this approach was proposed in the context of manually deriving the inverse for small programs, and we believe it is unlikely to scale to larger programs or to be amenable to automation. The second are grammar-based strategies that show that if the output of the original can be parsed using a deterministic grammar, then that approximates the original computation and can be used to derive the inverse [119, 270]. The limitation of this technique is that grammar-based approaches need to work with unambiguous, decidable grammars, which for all but the most trivial benchmarks is not possible.

Automatic programming The artificial intelligence community has explored *automatic programming* which resembles program synthesis. Approaches to automatic programming typically do not attempt to *generate* the program, but rather assemble it intelligently using already-existing components. Systems that follow a deductive methodology to such assembly include a genetic programming-based approach for composing abstract components using views (mappings between concrete types and abstract types) [212], an approach constructing astronomical data-manipulating programs [249], and even question answering [262], all reusing underlying domain-specific components. Systems also exist that follow a more inductive approach by generalizing from input-output examples [187, 79]. These are a natural fit for the kinds of techniques, e.g., those that infer explanations for a given set of data points, available in machine learning and the artificial intelligence community. Systems in this category include tools that can synthesize certain LISP programs [251], language-independent extensions [167, 236, 166], and logic programs [115, 113].

Simultaneous proof and program refinement When we fail to prove a property for a given program under a given abstraction, we refine the abstraction and try again, e.g., in model checking using counterexample guided abstraction refinement (CEGAR) [59, 57], or the same done lazily [148], or in an abstract interpretation framework [133]. Vechev, Yahav, and Yorsh propose an approach that refines the program in addition to the proof to synthesize both simultaneously [258]. They address the problem in the context of synthesizing synchronization, but the idea has applicability to general synthesis as well. While promising, refining the program simultaneously has the disadvantage of removing the monotonic progression that proof refinement implicitly contains. A careful choice is required in picking whether to refine the abstraction or the program when the verification fails for the current program and abstraction.

Synthesizing concurrency Concurrent programs are notoriously hard to design, and thus are a very promising target for automatic synthesis. Clarke and Emerson's seminal work on model checking was in fact proposed as a means of synthesizing synchronization skeletons [58]. From the same community, Pnueli and Rosner also addressed the problem of synthesizing distributed reactive systems from LTL specifications [215].

Vechev et. al. developed CGCExplorer [260, 261] for automatically exploring the space of concurrent garbage collectors and automatically synthesizing provably correct versions. They later extended it to a system called Paraglide for general synthesis [259]. Paraglide utilizes a model checker to validate candidate programs, much like the counterexample-guided inductive synthesis solution strategy for Sketching by Solar-Lezama et. al. [246]. Notably, Solar-Lezama's work also addresses the problem of synthesizing concurrent data structures.

Chapter 9

Conclusion

"What is the use of a new-born infant?"

— Benjamin Franklin[1]

We set out to show that we can build expressive and efficient techniques for program reasoning and program synthesis by encoding the underlying inference tasks as solutions to satisfiability instances. Reducing these problems to satisfiability allows us to leverage the engineering advances in current SAT and SMT solvers to build powerful program reasoning and synthesis tools. We have shown that it is possible to restrict attention to particular classes of proofs and programs (through templates) and to be able to automatically reason about and synthesize programs in those restricted classes.

We described algorithms that can reduce programming analysis problems to satisfiability instances over linear arithmetic and predicate abstraction. We have shown that using a satisfiability-based approach we can infer not only expressive

[1] When asked what was the use of a balloon, while he was the American Plenipotentiary to France; early 1780s.

invariants for verification, but also weakest pre- and strongest postconditions. Being able to infer expressive invariants will allow developers to build certifiably correct software. Being able to infer pre- and postconditions will allow developers to able to use and provide formal specifications of their software.

We have also shown how program synthesis can be viewed as generalized verification, allowing us to use the verifiers we developed for reasoning as synthesizers. We introduced the notion of a scaffold as a synthesis specification from which novel programs can be synthesized. A scaffold specifies a program as a template of its control flow, domain of expressions that appear in the program, and constraints on resources available. Using this approach, we envision that developers can delegate the task of building critical fragments of their codebases to a synthesizer that will automatically generate verified fragments that are guaranteed to be correct.

Lastly, we also showed how to construct a synthesizer that is inspired by testing. We leverage the core solving technology we developed for reasoning, and using symbolic traces as proxies for verification conditions, we show that we can synthesize programs by exploring a sufficient number of relevant paths through a template program. Just as we can view testing as an approximation to formal verification, this pragmatic synthesis approach can be viewed as using symbolic testing to generate programs with approximate guarantees.

Going forward, we envision that we can build on the foundations laid in this dissertation to develop techniques that can make programming easier, if not virtually redundant. Programming will be made *easier* by automatic and mechanized reasoning about programs. Tools will be able to automatically verify the correctness of programs, and for erroneous programs give the programmer the weakest conditions under which it fails. These tools will be able to automatically infer relevant pre- and postconditions that can be used as specifications or interfaces against which other components can be built. The task of programming will be *reduced* through automatic program synthesis. Programmers will write only part of the software, while the system will generate the provably-correct completion. Additionally, automatic program synthesis also holds the potential to generate new and novel algorithms.

Appendix A

Correctness of Satisfiability-based Algorithms

A.1 Linear Arithmetic: Correctness of Precondition Inference

Lemma A.1 ($\mathtt{N}_c$ = Immediately weaker neighbors) *For all relations I' that are weaker than I, there is some relation $I'' \in \mathtt{N}_c(I)$ such that $I \Rightarrow I'' \Rightarrow I'$.*

Proof: Suppose not, i.e., $\exists I'$ weaker than I such that $\not\exists I'' \in \mathtt{N}_c(I)$ such that $I \Rightarrow I'' \Rightarrow I'$. We first assume that the number of non-redundant conjuncts in both I and I' is the same. This assumption is valid because only a finite number

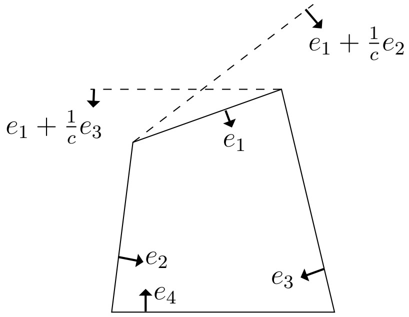


Figure A.1: Importance of staying within templates

of conjuncts (specified by the template) are permitted for the relations in the system. Otherwise it is possible to go a more expressive domain of relations and obtain weaker relations. Such an example is shown in Figure A.1: If both $e_1 + \frac{1}{c}e_3$ and $e_1 + \frac{1}{c}e_2$ can be added to the system then certainly a weaker relation can be constructed for which there is no element in $\mathtt{N}_c$ that is strictly stronger.

Then, without loss of generality, we can assume that I' is weaker only in the first conjunct (because such a relation is stronger than others with more conjuncts weaker than the corresponding conjunct in I). Thus I'' is obtained by weakening the first conjunct $e_1 \geq 0$ in I by a small amount. This can be done in two ways: either by adding an infinitesimally small constant δ to e_1 or by rotating e_1 by an infinitesimally small amount δ along the intersection of e_1 and e_l. By assumption we know that $\not\exists I'' \in \mathtt{N}_c(I)$ such that $I'' \Rightarrow I'$, and if I' is obtained by adding a small constant, then $\delta < \frac{1}{c}$, which leads to a contradiction since $\frac{1}{c}$ is the smallest constant expressible in the system. On the other hand, if I' is obtained by an infinitesimally small rotation then the smallest rotation possible is $\lim_{\epsilon \longrightarrow 0} e_1 + \epsilon e_l$, which we approximate by $e_1 + \frac{1}{c}e_l$. Again, if the rotation by δ is smaller, then $\delta < \frac{1}{c}$, which again leads to a contradiction.

□

Using the proof neighborhood $\mathtt{N}$ we will prove that, for program points not inside loops, the maximally weak preconditions (i.e., pointwise-weakest relations) for straight line fragments can be computed without iteration using locally pointwise-weakest relations. The proof makes use of a notion of the consistency of a relation with respect to certain others, as we defined below.

Definition A.1 **($(I_1, \ldots, I_k)$-consistent)** *A relation I, which is a conjunction of inequalities, is called $(I_1, \ldots, I_k)$-consistent if $I \Rightarrow I_i$ (or equivalently, $I \wedge \neg I_i$ is unsatisfiable) for all $1 \leq i \leq k$.*

Now we relate the definition above to the proof neighborhood $\mathtt{N}_c$ in the following lemma, and use it to connect pointwise-weakest relations and locally pointwise-relations in the theorem that follows.

Lemma A.2 *Let $I_1, \ldots, I_m$ be some given conjunctions of inequalities. Let I be some conjunction of inequalities that is $(I_1, \ldots, I_m)$-consistent. I is the weakest conjunctive relation that is $(I_1, \ldots, I_m)$-consistent iff for all $I'' \in \mathtt{N}_c(I)$, it is the case that $I'' \wedge \neg I_j$ is satisfiable for some $1 \leq j \leq m$.*

PROOF: The forward direction of the lemma is trivial. If I is the weakest relation that is $(I_1, \ldots, I_m)$-consistent then there cannot exist a strictly weaker relation I'' that is $(I_1, \ldots, I_m)$-consistent. Since all $I'' \in \mathtt{N}_c(I)$ are strictly weaker it has to be the case that $I'' \wedge \neg I_j$ is satisfiable for some $1 \leq j \leq m$.

We now show the reverse the direction of the lemma. From Lemma A.1 we know that for all relations I' weaker than I it is the case that $\exists I'' \in \mathtt{N}_c(I)$ such that $I \Rightarrow I'' \Rightarrow I'$. Let I' be the given weaker relation under consideration, and let I'' be a relation in $\mathtt{N}_c(I)$ such that $I'' \Rightarrow I'$. Also, let u be the index for which $I'' \wedge \neg I_u$ is satisfiable. Since $I'' \Rightarrow I'$ it has to be the case that $I' \wedge \neg I_u$ is also satisfiable. And therefore the weaker relation I' is not $(I_1, \ldots, I_m)$-consistent.

□

The neighborhood structure $\mathtt{N}_c$ has the following interesting property, which implies that no iteration is required for obtaining a weakest relation at a cut-point that lies outside any loop.

Theorem A.1 *Let π be a program point that does not lie inside any loop. Then, any locally pointwise-weakest relation (with respect to the neighborhood structure $\mathtt{N}_c$) at π is also a pointwise-weakest relation at π.*

Proof: Let I be a locally pointwise-weakest relation with respect to $\mathtt{N}_c$ at π. Let m be the number of paths to successor cut-points of π and let the weakest preconditions of the paths (as defined in Section 2.2.1 for paths) corresponding to them be $I_1, \ldots, I_m$ (i.e., $\omega(p_{i,j}, I_{\pi_j})$, where π_j is the jth successor cut-point and $p_{i,j}$ is the ith path connecting π and π_j). The program verification condition (Eq. 2.1) dictates that $I \Rightarrow I_i$ for all $1 \leq i \leq m$, i.e., I is $(I_1, \ldots, I_m)$-consistent. If I is also locally pointwise-weakest then that means that for all $I'' \in \mathtt{N}_c(I)$ it is the case that $I'' \wedge \neg I_j$ for some $1 \leq j \leq m$. Therefore, from Lemma A.2, we know that I is also the weakest relation that is $(I_1, \ldots, I_m)$-consistent, which implies that I is a pointwise-weakest relation at π.

□

Geometric Interpretation Lemma A.2 and Theorem A.1, and their proofs, have a nice geometric interpretation. The task of finding a pointwise-weakest relation I at a program point outside any loop can be shown equivalent to the task of finding the union of disjoint maximal convex regions that do not intersect with a given set of convex regions. Lemma A.2 implies that any convex region that does not intersect with a given set of convex regions is maximal iff moving any of its hyper-planes leads to an intersection with one of the convex regions from the given set. The interesting moves of a hyperplane involve either translation parallel to itself, or rotation along the intersection with another hyper-plane.

A.2 Linear Arithmetic: Refined neighborhood structure $\mathtt{N}_{c,\pi}$

The neighborhood structure $\mathtt{N}_c$ defined in Section 2.4.1, and used above, works well in practice. For sake of completeness we describe below a refined neighborhood structure $\mathtt{N}_{c,\pi}$ that works better in some cases.

Refined neighborhood structure $\mathtt{N}_{c,\pi}$ The neighborhood structure $\mathtt{N}_c$ defined above works well for two cases: (a) deducing pointwise-weakest relations at cut-points that are not inside any loop (b) deducing pointwise-weakest relations in which the inequalities are *independent* of each other, i.e., a small change in one of the inequalities does not require a change in any other inequality for the relation to remain consistent. The two cases described above cover the majority of cases in practice. In particular, they apply to the difficult benchmarks we experimented over, and also to the independently inductive inequalities addressed in previous work (e.g., [62, 219]).

However, for sake of completeness, we describe another neighborhood structure $\mathtt{N}_{c,\pi}$ that works better for cases other than (a) or (b).

We have already seen an example that violates (a) in Figure 2.7. The presence of the local minima forces us to iterate to obtain the global weakest precondition. An example of case (b) requires that the relation have inequalities that are dependent on each other. For instance, this would be the case when an equality expression $x = c$ is represented in terms of two inequalities $(x \leq c) \wedge (x \geq c)$. The neighborhood structure $\mathtt{N}_{c,\pi}$ is a refinement of $\mathtt{N}_c$, i.e., $\mathtt{N}_{c,\pi}$ reduces to $\mathtt{N}_c$ for the two cases described above.

For any relation I, $\mathtt{N}_c(I)$ includes all relations that are obtained from I by a small weakening of one of its inequalities. In contrast, the $\mathtt{N}_{c,\pi}(I)$ includes all those inequalities that are obtained from I by a small weakening of one of the independent inequalities and an appropriate weakening of the dependent inequalities. Since we do not know what these dependences are, one way to construct such neighbors is to find a satisfying solution to the original system of constraints in which the independent inequality is weakened slightly, and the independent unknown constants are forced to be same as before. An unknown constant d in a template relation I is dependent on an inequality in I at a program point π if changing the inequality in (any consistent solution to) I requires changing the constant d to obtain another consistent solution.

In practice, use of neighborhood structure $\mathtt{N}_{c,\pi}$ requires a small constant number of iterations to obtain a pointwise-weakest relation by iterating over locally pointwise-weakest relations.

A.3 Predicate Abstraction: Correctness of Optimal Solution Computation

Definition A.2 (Negatively-optimal solution) *Let ϕ be a formula with both positive and negative unknowns. Let P and N denote the set of positive and negative variables in ϕ, respectively, and let $S|_P$ and $S|_N$ denote the restriction of the solution to the positive and negative maps, respectively. Then a solution S is* negatively-optimal *if $S|_N$ is an optimal solution for $\phi[S|_P]$.*

Lemma A.3 (Modifying solutions (A)) *If S is a solution to a formula ϕ, then so is S', where S' is obtained from S by either taking a subsets of the positive assignments, or supersets of the negative assignments. Formally, let V be the set of all unknown in ϕ, and let S be a solution to ϕ. Then S' is also a solution if it is the case that $\forall_{\rho_i \in V}\ S'[\rho_i] \subseteq S[\rho_i] \wedge \forall_{\eta_i \in V}\ S'[\eta_i] \supseteq S[\eta_i]$.*

Proof: The proof follows directly from the definition of positive and negative variables in a formula ϕ. In particular, recall that if v is a positive unknown in ϕ and let $Q_1, Q_2 \subseteq Q(v)$, then

$$\forall S, Q_1, Q_2 : (Q_1 \Rightarrow Q_2) \quad \Rightarrow \quad (\phi S[v \mapsto Q_1] \Rightarrow \phi S[v \mapsto Q_2])$$

For the purposes of this lemma, we have Q_1 is $S[\rho]$ and Q_2 is $S'[\rho]$, i.e., we have $S'[\rho] \subseteq S[\rho]$ and so $Q_1 \Rightarrow Q_2$. Therefore, we know that $\phi X[S[\rho]] \Rightarrow \phi X[S'[\rho]]$ for any positive unknown ρ, and where X is an assignment to the remaining unknowns. By a similar argument, we know that $\phi X'[S[\eta]] \Rightarrow \phi X'[S'[\eta]]$ for any negative unknown η. This means that $\phi[S] \Rightarrow \phi[S']$. Since S and S' map each unknown variable to a predicate set, and from the definition of S being a solution, we know that $\phi[S]$ is *true*. Then for the implication to hold, we have $\phi[S']$ is *true* too.

□

Lemma A.4 (Modifying solutions (B)) *Let S^- be a negatively-optimal solution for ϕ. Let S^-_{extra} be identical to S^- except that $S^-[\rho] \subseteq S^-_{extra}[\rho]$ for some positive unknown ρ. Then if S^-_{extra} is also a solution to ϕ, then S^-_{extra} is negatively-optimal too.*

Proof: Again, from the definition of a positive variable ρ, we know that for $Q_1, Q_2 \subseteq Q(v)$

$$\forall S, Q_1, Q_2 : (Q_1 \Rightarrow Q_2) \quad \Rightarrow \quad (\phi S[v \mapsto Q_1] \Rightarrow \phi S[v \mapsto Q_2])$$

For the purposes of this lemma, we have Q_1 is $S^-_{extra}[\rho]$ and Q_2 is $S^-[\rho]$. Therefore, we know that $\phi X[S^-_{extra}[\rho]] \Rightarrow \phi X[S^-[\rho]]$, where X is an assignment to the remaining unknowns as before. Since all the other positive unknowns are identically assigned, we have that $\phi X'[S^-_{extra}|_P] \Rightarrow \phi X'[S^-|_P]$, where X' is some assignment to the negative unknowns. But we know that S^- is negatively-optimal for ϕ, i.e., X' is optimal for $\phi[S^-|_P]$, which by definition means that removing any predicate from any of the maps in X' makes S^- not a solution. (Note that S^- is $X' \cup S^-|_P$.) It may very well be that S^-_{extra} is not a solution, but if it is then for any X'' that is strictly weaker than X' leads to $\phi X''[S^-|_P]$ being false. Then because of the implication we just derived, it also means that X'' is not a solution for $\phi[S^-_{extra}|_P]$. Consequently, S^-_{extra} $(= X' \cup S^-_{extra}|_N)$ is also negatively-optimal.

□

We first prove a few auxiliary lemmas about the properties of `Merge` and `MakeOptimal`. Implicit in the definitions of `Merge` and `MakeOptimal` is the assumption that right before returning the sanitize their solutions, i.e., add any predicate from $Q(\rho)$ that is implied by $\sigma[\rho]$ and removing any predicate from $\sigma[\eta]$ that is implied by the remaining. This allows us to treat superset as the implication relation, and treat predicates as independent of each other. We assume that the predicate sets contain at least one *true* for positives, essentially the empty set, and they contain $false$ for the negatives, or some set of predicates that can imply $false$.

Consider a formula ϕ and its positive and negative unknowns. Each of the positive unknowns defines its own space, and each predicate assignment to the unknown defines a half-hyperplane in that space. The set S (Line 8 in `OptimalSolutions`) as constructed, contains for all possible single hyperplane combinations (one from each space) the weakest assignments to the negatives (i.e, negatively optimal). Let us call each of the elements of S a *basis*.

Definition A.3 (Basis set) *Given a map $\sigma = \{\rho_i \mapsto Q_i\}_{i=1..n} \cup \{\eta_i \mapsto Q_i\}_{i=1..m}$, let us call $\sigma|_{+ve}$ as the first set of maps (for the positive unknowns) and $\sigma|_{-ve}$ the second set of maps (for the negative unknowns).*

Let $C = \sigma[\rho_1] \times \sigma[\rho_2] \times .. \times \sigma[\rho_n]$ denote the set of all combinations of the positive maps. We call a collection of basis elements $X (\subset S)$ a basis set *for σ, if* *for each $c \in C$, the map formed by c augmented with $\sigma|_{-ve}$ has an element in $x \in X$ such that (1) $x|_{+ve} = c|_{+ve}$, and (2) $c|_{-ve} \Rightarrow x|_{-ve}$.*

Lemma A.5 *Every solution has a basis set.*

Proof: Let σ be the solution. Consider the combinations $\{c_i\}_i$ of the positives $\sigma|_{+ve}$. Since each is a pointwise subset of $\sigma|_{+ve}$, by Lemma A.3, we know that each combination (with identical negatives), i.e., $\sigma_i (= c_i \cup \sigma|_{-ve})$, is also a solution. Now consider an individual σ_i and its positives $\sigma_i|_{+ve}$. From the property of `OptimalNegativeSolutions` (Corollary A.1) in constructing negatively-optimal solutions, we know that the negatives $\sigma_i|_{-ve}$ of the solution have to be strictly stronger, i.e., a superset, of the basis with the positives equal to $\sigma|_{+ve}$. Therefore, σ has a basis set.

□

The reverse, that a set of basis elements can be lifted to a solution, also holds.

Lemma A.6 (Lifting basis elements) *A map σ is a solution if it has a basis set.*

Proof: Let $X (\subseteq S)$ be a set of basis elements. We will show that $\sigma' \doteq \uplus_{x \in X} x$ is a solution. Then if X is a basis set for σ, then by Lemma A.5 we know that σ is just σ' with additional elements in the negatives. Then, by Lemma A.3 we know that if σ' is a solution, then so is σ.

To show that $\sigma' \doteq \uplus_{x \in X} x$ is a solution, we present a geometric proof. Consider the assignment $pos_1 \doteq \{\rho_1 \mapsto \{q\}, \rho_2 \mapsto \{q'\}, .., \rho_n \mapsto \{q''\}\}$, where

$q \in Q(\rho_1), q' \in Q(\rho_2), .., q'' \in Q(\rho_n)$ to the positive unknowns in a basis element x. This assignment defines a half-space in an n-dimensional space. Each positive unknown defines a dimension and a predicate induces a half-space. Let us say that pos_2 is another assignment to the positives. Their disjoint union $pos_1 \uplus pos_2$ corresponds to the intersection of the half-spaces. Corresponding to each of pos_1 and pos_2 we have negatively-optimal solutions neg_1 and neg_2, respectively, that themselves define half-spaces in the dimensions defined by the negative unknowns. We now compare the negative solutions for the formulae $\phi[pos_1]$ and $\phi[pos_1 \uplus pos_2]$, where ϕ is the original formula. It has to be the case that for comparable solutions the negatively-optimal solutions to $\phi[pos_1 \uplus pos_2]$ are strictly stronger than $\phi[pos_1]$ (and also $\phi[pos_2]$). In particular, one solution to the negatives in $\phi[pos_1 \uplus pos_2]$ would be $neg_1 \uplus neg_2$. By induction, this argument generalizes to disjoint unions of multiple solutions the result of which is guaranteed to be a solution.

□

Lemma A.7 (`Merge`) *The procedure* `Merge` *returns the join $\sigma_1 \uplus \sigma_2$ of two maps σ_1 and σ_2, if the join is a valid solution, else it indicates failure by returning $\perp$. Here $\uplus$ indicates the piecewise union of two maps.*

Proof: The first part, i.e, it return the join $\uplus$ if it does not fail, is trivial from the definition of the procedure. We just need to show that if it does not fail, then the returned value is a valid solution.

A corollary to Lemma A.3 is that, compared to a solution X, any X' that is weaker in the positive or stronger in the negatives is also a solution. (By simple translation of the superset relation to implication.) The set T is a restriction of the basis set to those whose negatives are weaker than the current join. Thus since T contains only those basis whose negatives are weaker, the X' we have is stronger and will be a solution if the positives are kept unchanged.

Lastly, checking individually for positives, within T (which guarantees solutions consistent for the negatives), we make sure that *every* combination of positives had a valid negative map, ensuring that their accumulation is also a valid solution (Lemma A.5).

□

Lemma A.8 *Let σ' be in S with $\sigma'|_{+ve} = \{\rho_k \mapsto p\} \cup \{\rho_i \mapsto \{true\}\}_{i \neq k}$ and $\sigma \uplus \{\rho_k \mapsto \{p\}\}$ is a solution, and $\sigma|_{-ve} \Rightarrow \sigma'|_{-ve}$, then calling the procedure* `Merge` *with σ, σ' and S does not fail.*

Proof: From $\sigma|_{-ve} \Rightarrow \sigma'|_{-ve}$ we know that $(\sigma \uplus \sigma')|_{-ve} \Rightarrow \sigma'|_{-ve}$. We also have from assumption that $\sigma' \in S$ and therefore σ' is in T (Line 3). In the join, the positives are $\sigma|_{+ve} \uplus \{\rho_k \mapsto \{p\}$ and the negatives are exactly as strong as $\sigma|_{-ve}$.

Because $\sigma \uplus \{\rho_k \mapsto \{p\}\}$ is a solution, we know that some basis set exists for the enumerated combinations of the positives, and hence the conditional on Line 4 evaluates to *true*. Therefore the procedure does not fail (Lemma A.5).

□

Lemma A.9 *If $\sigma \uplus \{\rho_k \mapsto \{p\}\}$ is a solution to ϕ (whose negative unknowns are N), then the negatively-optimal solution to $\phi[\rho_k \mapsto \{p\}][\rho_i \mapsto \{true\}]_{i \neq k}$ is a subset of $\sigma|_N$.*

Proof: Again, from the definition of a positive variable ρ, we know that for $Q_1, Q_2 \subseteq Q(\rho)$

$$\forall S, Q_1, Q_2 : (Q_1 \Rightarrow Q_2) \quad \Rightarrow \quad (\phi S[\rho \mapsto Q_1] \Rightarrow \phi S[\rho \mapsto Q_2])$$

For the purposes of this lemma, we have Q_1 is $\sigma|_P \uplus \{\rho_k \mapsto \{p\}\}$ and Q_2 is $\{\rho_i \mapsto \{true\}\}_{i \neq k} \cup \{\rho_k \mapsto \{p\}\}$. Therefore, we know that $\phi X[\sigma|_P \uplus \{\rho_k \mapsto \{p\}\}] \Rightarrow \phi X[\{\rho_i \mapsto \{true\}\}_{i \neq k} \cup \{\rho_k \mapsto \{p\}\}]$, where X is an assignment to the remaining (negative) unknowns. If X is the negatively-optimal solution, then the consequent of the implication is $true$ under it and for every $X' \subset X$ (pairwise subset) is it $false$. That implies that for every X' that is a subset the antecedent also has to be $false$, i.e., it would not form a valid solution. Therefore $\sigma|_N$ has to be a superset of the negatively-optimal solution X.

□

Lemma A.10 (`MakeOptimal`) *The* `MakeOptimal` *procedure has the property that corresponding to a negatively-optimal σ, the procedure returns an optimal solution.*

Proof: We will show that three invariants hold about the loop from Lines 2–4 in `MakeOptimal`: (1) no extraneous predicates are added to the negative solutions, i.e., the negative solutions remain maximally-weak, (2) σ is a solution in every iteration, and (3) on termination, there is no predicate that can be added to σ while still ensuring that it is a solution. Using invariants (1) and (2) and Lemma A.4 we get the additional invariant that the solution σ is negatively-optimal in every iteration. Adding (3), we get that, at termination, the solution is also optimal.

We now show that the three properties hold of the loop. For (1), notice that the loop only calls `Merge` with an element from set T, which in turn only contains solutions that are pointwise, at all negative unknowns, weaker than σ. Therefore, the join $\uplus$ of a set weaker than itself, yields the same set, and therefore the negatives remain maximally-weak. For (2), notice that the loop leaves σ unchanged if the merge failed, which happens if the merged result is not a solution (Lemma A.7), and therefore σ is only updated with valid solutions.

For (3), we need a little bit more effort. Suppose there exists a predicate p, *not already there*, that can be added to some positive unknown ρ_k's map, while the result $\sigma \uplus \{\rho_k \mapsto \{p\}\}$ still being a solution. If that is the case, then by Lemma A.3 we know that $N \uplus P$ is also a solution, where P is $\{\rho_k \mapsto \{p\}\} \uplus \{\rho_i \mapsto \{true\}\}_{i \neq k}$, and N is σ but restricted to the negative unknowns. ($true$ is equivalent to the empty set, i.e., a subset of every set.) Also, let N_{start} be σ at the start of the loop restricted to the negative unknowns. Note that by (1), N is neither weaker or stronger than N_{start}.

Now notice that the negatively-optimal map N' corresponding to $\phi[P]$ has to be a subset of N or else $\sigma \uplus \{\rho_k \mapsto \{p\}\}$ cannot be a solution (by Lemma A.9). Being a subset of N means that it is at least as weak as N. From the above observation about N_{start} it also means that N' is at least as weak as N_{start} too. If that is the case, then $N' \uplus P$ must have been in T and therefore must have been merged with σ at some point. Since the map for ρ_k does not contain p, it implies that the merge did not yield a valid solution. But this contradicts Lemma A.8, which states that a merge over σ' ($= N' \uplus P$ and $\in S$) and σ does not fail. Therefore, no such predicate can exist.

□

Before proving the general lemma about the correctness of `OptimalSolutions` (Lemma A.3), we prove a restricted version first. The theorems make use of the correctness of `OptimalNegativeSolutions` as described by Theorem A.13.

Theorem A.2 (Correctness of `OptimalSolutions` for restricted formulae) *Let ϕ be a formula with positive and negative unknowns with the positive and negative unknowns uncorrelated in the following manner. If S is an optimal solution to ϕ, then any S' with positive variables assigned subsets (compared to S's positives) is only a solution if the negatives are assigned supersets (as compared S's negatives).*

Let $\{v_i\}_i$ is the set of all unknown variables in ϕ and let $\mathbb{S}$ be the set of all possible assignments to v_i's, i.e., $2^{Q(v_1)} \times 2^{Q(v_2)} \times 2^{Q(v_n)}$. Then the procedure `OptimalSolutions`(ϕ, Q) *returns the set*

$$\{S \mid S \in \mathbb{S} \textit{ and } S \textit{ is an optimal solution for } \phi \textit{ with respect to } Q\}$$

Proof: For the sake of brevity in the proof, we assume that ϕ contains one positive ρ and one negative unknown η. The proof works exactly as is for the case of multiples, with required conjunctions, unions, added in appropriate places. Also, let us use the notation $\left\{ \begin{array}{c} p_1 \mathinner{.\,.} p_n \\ \\ q_1 \mathinner{.\,.} q_m \end{array} \right\}$ to denote the solution map $\{\rho \mapsto \{p_1 \mathinner{.\,.} p_n\}, \eta \mapsto \{q_1 \mathinner{.\,.} q_m\}\}$, where each $p_i \in Q(\rho)$ and each $q_i \in Q(\eta)$. We prove that for a solution S is in the output set of `OptimalSolutions` iff it is optimal. We prove each direction in turn:

"⇒" From Corollary A.1 (described later), we know that the calls to the procedure `OptimalNegativeSolutions` produce negatively-optimal solutions. From the optimality of the output values of `MakeOptimal` (Lemma A.10), all solutions in R after Line 8 are optimal. The only other additions to R are again outputs of `MakeOptimal` (added through the call to `Saturate` on Line 9), and consequently, at the end R only contains solutions that are optimal.

"⇐" Let $\left\{ \begin{array}{c} p_1 \mathinner{.\,.} p_n \\ \\ q_1 \mathinner{.\,.} q_m \end{array} \right\}$ be the optimal solution to ϕ. Then we know from Lemma A.3 that $\left\{ \begin{array}{c} p_1 \\ \\ q_1 \mathinner{.\,.} q_m, \mathinner{.\,.}, q'_m \end{array} \right\}, \left\{ \begin{array}{c} p_2 \\ \\ q_1 \mathinner{.\,.} q_m, \mathinner{.\,.}, q''_m \end{array} \right\}, \mathinner{.\,.}, \left\{ \begin{array}{c} p_n \\ \\ q_1 \mathinner{.\,.} q_m, \mathinner{.\,.}, q'''_m \end{array} \right\}$ are therefore all solution too (not optimal though), where each set of assignments to the negatives is a superset as indicated by the $q'_m, q''_m, \mathinner{.\,.}, q'''_m$. Line 6 in the procedure accumulates optimal negative solutions for individual predicates $p_1, p_2, \mathinner{.\,.}, p_n$. From Lemma A.13 (correctness of `OptimalNegativeSolutions`), we know that the outputs will be the minimal sets to the negative unknown.

By virtue of $\left\{\begin{array}{c} p_1 \cdot\cdot p_n \\ \\ q_1 \cdot\cdot q_m \end{array}\right\}$ being an optimal solution and the uncorrelated ϕ we consider in this theorem, this means that the output at Line 6 will be exactly $\left\{\begin{array}{c} p_i \\ \\ q_1 \cdot\cdot q_m \end{array}\right\}$.

That means that all of $\left\{\begin{array}{c} p_1 \\ \\ q_1 \cdot\cdot q_m \end{array}\right\}, \left\{\begin{array}{c} p_2 \\ \\ q_1 \cdot\cdot q_m \end{array}\right\}, \ldots, \left\{\begin{array}{c} p_n \\ \\ q_1 \cdot\cdot q_m \end{array}\right\}$ are in the solution set S right before line 8 in `OptimalSolutions`. From Lemma A.10, this implies that each one of these elements in S will be lifted to $\left\{\begin{array}{c} p_1 \cdot\cdot p_n \\ \\ q_1 \cdot\cdot q_m \end{array}\right\}$.

Therefore the set R will contain $\left\{\begin{array}{c} p_1 \cdot\cdot p_n \\ \\ q_1 \cdot\cdot q_m \end{array}\right\}$ after Line 8. Since the procedure `Saturate` (called on Line 9) does not delete elements from R, this solution will be in the output of the procedure.

□

Lemma A.11 *Let S, S' be optimal solutions. The following hold separately: (a) if $S|_{+ve} \supseteq S'|_{+ve}$, then $S|_{-ve} \not\subseteq S'|_{-ve}$; (b) if $S|_{-ve} \subseteq S'|_{-ve}$, then $S|_{+ve} \not\supseteq S'|_{+ve}$;*

Proof: Both cases are similar and straightforward:

(a) Suppose not, i.e., $S|_{-ve} \subseteq S'|_{-ve}$. From Lemma A.3 we know that $S|_{+ve} \cup S'|_{-ve}$ is a solution (as we are just adding some predicates to some negative assignment in the solution $S = S|_{+ve} \cup S|_{-ve}$). But this contradicts the optimality of $S'|_{+ve} \cup S'|_{-ve}$, i.e., that $S'|_{+ve}$ contains as many predicates as possible.

(b) Suppose not, i.e., $S|_{+ve} \supseteq S'|_{+ve}$. From Lemma A.3 we know that $S'|_{+ve} \cup S|_{-ve}$ is a solution (as we are just removing some predicates from some positive assignment in the solution $S = S|_{+ve} \cup S|_{-ve}$). But this contradicts the optimality of $S'|_{+ve} \cup S'|_{-ve}$, i.e., that $S'|_{-ve}$ contains as few predicates as possible.

□

Claim A.1 (Every solution can be split on the negatives) *If σ is a solution then there exist σ_1, σ_2 solutions that are decompositions of σ, i.e., $\sigma_1|_{-ve} \cup \sigma_2|_{-ve} = \sigma|_{-ve}$ and $\sigma_1|_{+ve} \cup \sigma_2|_{+ve} \supseteq \sigma|_{+ve}$.*

Proof: We present a geometric proof as we did for Lemma A.6. Consider the assignment $neg_1 \doteq \{\eta_1 \mapsto \{q_{11}, q_{12}, ..\}, \eta_2 \mapsto \{q_{21}, q_{22}, ..\}, .., \eta_n \mapsto \{q_{n1}, q_{n2}, ..\}\}$, where $q_{ij} \in Q(\eta_i)$, to the negative unknowns. This assignment defines an intersection of half-spaces in an n-dimensional space. Each negative unknown defines a dimension and a predicate induces a half-space. Multiple predicates induces an intersection of half-spaces. Let neg_2 is the other assignment to the negatives. Their disjoint union $neg_1 \uplus neg_2$ corresponds to the intersection of the half-spaces. Corresponding to each of neg_1 and neg_2 we have optimal solutions pos_1 and pos_2, respectively, that themselves define half-spaces in the dimensions defined by the positive unknowns.

We now compare the optimal positive solutions for the formulae $\phi[neg_1]$ and $\phi[neg_1 \uplus neg_2]$, where ϕ is the original formula. It has to be the case that for comparable solutions the optimal solutions to $\phi[neg_1 \uplus neg_2]$ are strictly stronger

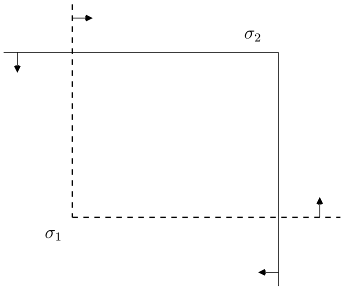


Figure A.2: Illustrating the decomposition of the negative solution.

than $\phi[neg_1]$ (and also $\phi[neg_2]$). In particular, one solution to the negatives in $\phi[neg_1 \uplus neg_2]$ would be $pos_1 \uplus pos_2$. Additionally, since any subset of the positives is also a solution, by Lemma A.3, we have that $\sigma_1|_{+ve} \uplus \sigma_2|_{+ve} \supseteq \sigma|_{+ve}$.

□

Example A.1 *It is instructive to consider an example formula $\phi \doteq \eta \Rightarrow \rho$. The one negative unknown η defines a dimension and assignments of predicates define subspaces in that dimension, as shown in Figure A.2. Now consider partial solution $\sigma \doteq \{\eta \mapsto \{-10 < x, x < 10, -5 < y, y < 5\}\}$, $\sigma_1 \doteq \{\eta \mapsto \{-10 < x, -5 < y\}\}$ and $\sigma_2 \doteq \{\eta \mapsto \{x < 10, y < 5\}\}$. Notice that the ($\rho$) solutions to $\phi[\sigma|_{-ve}]$ can include predicates implied by $-10 < x < 10 \wedge -5 < y < 5$, while those to $\phi[\sigma_1|_{-ve}]$ can only include those implied by $-10 < x \wedge -5 < y$. This entails that the former predicate map can be stronger than the latter.*

Lemma A.12 *Let $\left\{ \begin{array}{c} p_1 \mathinner{..} p_{s_1} \\ \\ q_1 \mathinner{..} q_{t_1} \end{array} \right\}$, $\left\{ \begin{array}{c} p_1 \mathinner{..} p_{s_2} \\ \\ q_1 \mathinner{..} q_{t_2} \end{array} \right\}$,.. be optimal solutions in R, with $\forall i : \{q_1, \mathinner{..}, q_{t_i}\} \subseteq \{q_1, \mathinner{..}, q_m\}$ and $\forall i : \{p_1, \mathinner{..}, p_{s_i}\} \subseteq \{p_1, \mathinner{..}, p_n\}$. Let $X = \left\{ \begin{array}{c} p_1 \mathinner{..} p_n \\ \\ q_1 \mathinner{..} q_m \end{array} \right\}$ also be an optimal solution, and let $\{p_1, \mathinner{..}, p_n\} = \cup_i \{p_1, \mathinner{..}, p_{s_i}\}$. Then $X \in \texttt{Saturate}(R, S)$.*

Proof: From Lemma A.7 we know that the procedure `Merge` returns the disjoint union $\sigma_1 \uplus \sigma_2$ of its argument solutions σ_1 and σ_2, if $\sigma_1 \uplus \sigma_2$ is indeed a valid solution. Therefore, we just need to show that there exists a decomposition of X as $(((\sigma_1 \uplus \sigma_2) \uplus \sigma_3) \,.\,. \uplus \sigma_n)$, such that each subexpression is a valid solution. (Technically, the decomposition is $\texttt{lift}(\texttt{lift}(\texttt{lift}(\sigma_1 \uplus \sigma_2) \uplus \sigma_3) \,.\,. \uplus \sigma_n)$, where `lift` indicates the augmenting of the positives in some σ to the optimal through a call to $\texttt{MakeOptimal}(\sigma, S)$. Additionally, we would need to worry about early termination of the outermost loop in `Saturate` on Line 1 and the conditional on Line 4. We defer these concerns until later.) This decomposition essentially means that there is a binary tree (of two way splits, on both the positives and negatives) such that every node in the tree is a valid solution.

We prove that such a binary tree exists by showing that every optimal solution can be decomposed into two solutions that are themselves optimal, i.e., for every σ there exists σ_1, σ_2 such that $\sigma = \sigma_1 \uplus \sigma_2$ and all three are optimal. Suppose such a decomposition is not possible. Since for the negatives any superset if always a solution, we consider the disjoint split of the predicates in σ_{-ve} into N_a and N_b. If a decomposition is not possible then that implies that the optimal positive solutions (which will have the maximal number of predicates they can have), corresponding to every set of N_a and N_b will not union up to σ_{+ve}. For that to be the case, all splits of σ_{-ve} into N_a and N_b, can have optimal positive solutions that at max union up to a subset of σ_{+ve}. But by Claim A.1, this means that σ cannot be a solution—contradiction.

□

Theorem A.3 (Correctness of `OptimalSolutions`) *Let $\{v_i\}_i$ is the set of all unknown variables in ϕ and let $\mathbb{S}$ be the set of all possible assignments to v_i's, i.e., $2^{Q(v_1)} \times 2^{Q(v_2)} \times 2^{Q(v_n)}$. Then the procedure* `OptimalSolutions`(ϕ, Q) *returns the set*

$$\{S \mid S \in \mathbb{S} \textit{ and } S \textit{ is an optimal solution for } \phi \textit{ with respect to } Q\}$$

Proof: We build on the proof for the restricted case (Theorem A.2). The proof of the forward "⇒" direction remains identical to the restricted case. The reverse "⇐" direction needs more work, since now the output at Line 6 may have solutions of the form $\left\{\begin{array}{c} p_i \\ \\ q_1 \,..\, q_t \end{array}\right\}$, and the following cases arise

- $\{q_1, .., q_t\} \subseteq \{q_1, .., q_m\}$: From Lemma A.10, we know that for each of the elements after Line 6, `MakeOptimal` returns an optimal solutions with the same negatives and augmented positives. Since $\left\{\begin{array}{c} p_i \\ \\ q_1 \,..\, q_t \end{array}\right\}$ is optimally-negative, from Lemma A.10, we know that `MakeOptimal` will lift each p_i to the maximal number of predicates $\{p_1 \,..\, p_s\}$ that can occur. Now, because both $\left\{\begin{array}{c} p_1 \,..\, p_s \\ \\ q_1 \,..\, q_t \end{array}\right\}$ and $\left\{\begin{array}{c} p_1 \,..\, p_n \\ \\ q_1 \,..\, q_m \end{array}\right\}$ are optimal solutions, by Lemma A.11 that $\{p_1 \,..\, p_s\} \subseteq \{p_1 \,..\, p_n\}$. Then by Lemma A.12, the theorem follows.

- $\{q_1, .., q_m\} \subseteq \{q_1, .., q_t\}$: By Lemma A.11 this case cannot arise as both $\left\{\begin{array}{c} p_i \\ \\ q_1 \,..\, q_t \end{array}\right\}$ and $\left\{\begin{array}{c} p_1 \,..\, p_n \\ \\ q_1 \,..\, q_m \end{array}\right\}$ are optimal solutions.

- $\{q_1, .., q_t\}$ is orthogonal to $\{q_1, .., q_m\}$: We leave this case as an exercise to the reader.

□

Lemma A.13 (Correctness of `OptimalNegativeSolutions`) *Let $\{\eta_i\}_i$ is the set of all unknown variables in ϕ^-, a formula that contains only negative unknowns, and let $\mathbb{S}^-$ be the set of all possible assignments to η_i's, i.e., $2^{Q(\eta_1)} \times 2^{Q(\eta_2)} \times 2^{Q(\eta_n)}$. Then the procedure* `OptimalNegativeSolutions`(ϕ^-, Q) *returns the set $\mathbb{S}^-_{opt} = \{S^- \mid S^- \in \mathbb{S}^-$ and S^- is an optimal solution for ϕ^- with respect to $Q\}$.*

Proof: The procedure `OptimalNegativeSolutions` searches top to bottom in a lattice ordered by the subset relation, i.e., with $S_1 \sqsubseteq S_2 \iff S_1 \supseteq S_2$. (This ordering is more intuitive using the implication relation, i.e. $S_1 \sqsubseteq S_2 \iff (\bigwedge_{s_1 \in S_1} s_1) \Rightarrow (\bigwedge_{s_2 \in S_2} s_2)$) We prove that a solution S^- is in the returned set for the procedure iff it is in $\mathbb{S}^-_{opt}$.

"$\Rightarrow$" By the enumeration over the lattice, i.e., construction, we know that the solution S^- output by the procedure has to be in $\mathbb{S}^-$. We just need to prove that it is optimal too. Suppose not, then a solution S^-_1 with assignments one of which is a strict subset is also a solution. Such a solution would be ordered above S^- in the lattice, i.e. $S^- \sqsubseteq S^-_1$. But since the procedure does a top to bottom search, it would have encountered S^-_1 and deleted its subtree if S^-_1 was found to be a solution. But since the subtree was not deleted (because an element, S^-, from the subtree was output), we conclude that S_1^- is not a solution. Contradiction.

"$\Leftarrow$" Since S^- is in $\mathbb{S}^-_{opt}$ we know that it is in $\mathbb{S}^-$ and is also optimal. It will be in the output of the procedure if every element on every path from it to the root ($\top$, i.e., the empty set) is not a solution, i.e., every element that is a strict subset is not a solution. From the definition of optimality, and that S^- is optimal, we know that to be true. Hence S^- is in the output of the procedure.

□

The following is a direct corollary of the above lemma.

Corollary A.1 (Producing negatively-optimal solutions) *A solution is in the output of* `OptimalNegativeSolutions` *iff it is negatively-optimal.*

A.4 Predicate Abstraction: Correctness of the Reduction to SAT

We first show the boolean encoding for each individual verification condition is sound. The proof relies on Lemma A.6 concerning the lifting of basis elements to solutions.

Lemma A.14 (Correctness of VC encoding) *An assignment that satisfies the boolean formula $\psi_{\delta,\tau_1,\tau_2,\sigma_t}$ (Eq. (3.7)) induces a map that is a solution to the verification condition corresponding to $\delta, \tau_1, \tau_2, \sigma_t$.*

Proof: Let S_{bool} be some satisfying assignment to the variables $b_q^{v_i}$ that appear in $\psi_{\delta,\tau_1,\tau_2,\sigma_t}$. Then we show that $S = \{v_i \mapsto \{q \mid q \in Q(v_i), S_{bool}[b_q^{v_i}] = true\}\}_i$ is a solution to, i.e., it satisfies, the corresponding VC formulae. From the assumption that the predicate map for every positive ρ_i contains the predicate *true*, the boolean assignment has at least one boolean $b_q^{\rho_i}$ assigned *true* for some q. Since Eq. (3.7) is satisfied, we know that for each of the combinations of the positives, the assignment has at least as many elements in the negatives such that the corresponding basis element is a solution. This means that the corresponding map has a basis set, and by Lemma A.6 we infer that the map is a solution to the verification condition.

□

Theorem A.4 (Correctness of SAT encoding) *The boolean formula $\psi_{\texttt{Prog}}$ (from Eq. (3.8)) is satisfiable iff there exists an invariant solution for program* `Prog` *over predicate-map Q.*

Proof: We prove each direction, of $\psi_{\texttt{Prog}}$ is satisfiable ⇔ invariant solution exists, in turn:

⇒ If $\psi_{\texttt{Prog}}$ is satisfiable that implies that each conjunct in Eq. (3.8) is satisfied by some assignment which in turn means that each conjunct in Eq. (3.7) is satisfied by the assignment. Let S_{bool} be some satisfying assignment to the variables $b_q^{v_i}$ that appear in $\psi_{\texttt{Prog}}$. Then we show that $S = \{v_i \mapsto \{q \mid q \in Q(v_i), S_{bool}[b_q^{v_i}] = true\}\}_i$ is an invariant solution, i.e., it satisfies each of the VC formulae. By Lemma A.14, we know that any satisfying solution to Eq. (3.7) induces a solution to the corresponding VC. Since S_{bool} simultaneously satisfies all clauses generated through Eq. 3.7, it induces a map that simultaneously a solution all VCs—therefore an invariant solution.

$\Leftarrow$ Let $S = \{v_i \mapsto Q_i\}_i$ be the invariant solution. Then we show that the map $S_{bool} = \{b_q^{v_i} \mapsto true \mid q \in Q_i\}_i \cup \{b_q^{v_i} \mapsto false \mid q \in Q(v_i) \setminus Q_i\}_i$ is a satisfying assignment to $\psi_{\texttt{Prog}}$. We show that S individually satisfies each conjunct in Eq. (3.8) which in turn means that it satisfies each conjunct in Eq. (3.7). From the presence of the predicate *true* in the predicate sets, we know that each positive is assigned some predicate by the invariant solution.

Since S is an invariant solution, it satisfies each of the verification conditions of the program. Consider the formula $\texttt{VC}(\langle\tau_1, \delta, \tau_2'\rangle)$. By Lemma A.5, we know that the solution S to the formula has a basis set. By Definition A.3 we have that the basis set contains elements whose positives are the (single-element) enumerations and the negatives are weaker than those of S. Each element of the basis set satisfies the VC formula as well. The implications in Eq. (3.7) encode exactly this basis set. It states that for each enumeration of the positives (antecedent), at least one of the optimally-negative solutions be valid (consequent). Thus for all positive enumerations in S the corresponding boolean indicators will be set to *true* and we know that at least one disjunct in the consequent will be *true* for the induced assignment.

□

Bibliography

[1] Phoenix. `http://research.microsoft.com/Phoenix/`.

[2] Modular verification of software components in c. *IEEE Trans. Softw. Eng.*, 30(6):388–402, 2004.

[3] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48, New York, NY, USA, 2007. ACM.

[4] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM.

[5] Andrew W. Appel. *Modern compiler implementation in ML/Java/C.* Cambridge University Press, New York, NY, USA, 1997.

[6] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[7] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *Proceedings of the 5th IFAC Cconference on System Structure and Control (SSC'98)*, pages 469–474. Elsevier Science, July 1998.

[8] David Aspinall and Marin Hofmann. Dependent types. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 45–86. MIT Press, 2005.

[9] Ittai Balaban, Ariel Cohen, and Amir Pnueli. Ranking abstraction of recursive programs. In *VMCAI*, pages 267–281, 2006.

[10] Thomas Ball, Byron Cook, Satyaki Das, and Sriram K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS*, pages 388–403, 2004.

[11] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, New York, NY, USA, 2001. ACM.

[12] Thomas Ball, Todd Millstein, and Sriram K. Rajamani. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst.*, 27(2):314–343, 2005.

[13] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, London, UK, 2001. Springer-Verlag.

[14] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130, London, UK, 2000. Springer-Verlag.

[15] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

[16] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs 0002, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[17] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS*, volume LNCS 3362. Springer, 2004.

[18] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In *17th International Conference on Computer Aided Verification*, pages 20–23. Springer, 2005.

[19] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in sat modulo theories. In M. Hermann and A. Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06), Phnom Penh, Cambodia*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer, 2006.

[20] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[21] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[22] D. Basin, Y. DeVille, P. Flener, A. Hamfelt, and J. F. NIlsson. Synthesis of programs in computational logic. In *Program Development in Computational Logic, Lecture Notes in Computer Science LNCS 3049*. Springer, 2004.

[23] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Variance analyses from invariance analyses. In *POPL*, pages 211–224, 2007.

[25] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[26] Ottmar Beucher. *MATLAB und Simulink (Scientific Computing)*. Pearson Studium, 08 2006.

[27] Dirk Beyer, Adam J. Chlipala, and Rupak Majumdar. Generating tests from counterexamples. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 326–335, Washington, DC, USA, 2004. IEEE Computer Society.

[28] Dirk Beyer, Thomas Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, volume 4349 of *LNCS*, pages 378–394, 2007.

[29] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007.

[30] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[31] Dirk Beyer, Tom Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, 2007.

[32] Nikolaj Bjørner and Joe Hendrix. Linear functional fixed-points. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 124–139, Berlin, Heidelberg, 2009. Springer-Verlag.

[33] 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 *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 2003. ACM Press.

[34] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical

real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation.*, LNCS 2566, pages 85–108. October 2002.

[35] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. In *International Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, pages 3–16, 2007.

[36] Roderick Bloem, Krishnendu Chatterjee, Thomas Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In Springer, editor, *Computer Aided Verification (CAV)*, pages 140–156, 2009.

[37] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *DATE*, pages 1188–1193, 2007.

[38] Roderick Bloem, Karin Greimel, Thomas Henzinger, and Barbara Jobstmann. Synthesizing robust systems. In *Conference on Formal Methods in Computer Aided Design (FMCAD'09)*, pages 85–92, 2009.

[39] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter Rossum, Stephan Schulz, and Roberto Sebastiani. Mathsat: Tight integration of sat and mathematical decision procedures. *Journal of Automated Reasoning*, 35(1-3):265–293, 2005.

[40] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *CAV*, pages 335–349, 2005.

[41] Aaron R. Bradley and Zohar Manna. Verification constraint problems with strengthening. In *ICTAC*, pages 35–49, 2006.

[42] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram K. Rajamani, editors, *Proc. 17th Intl. Conference on Computer Aided Verification (CAV)*, volume 3576 of *LNCS 3576*. Springer Verlag, July 2005.

[43] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *Proc. 32nd International Colloquium on Automata, Languages and Programming*, volume 3580 of *LNCS 3580*, pages 1349–1361. Springer Verlag, 2005.

[44] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What 's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation: 7th International Conference, (VMCAI)*, volume 3855, pages 427–442, Charleston, SC, January 2006. Springer Verlag.

[45] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. pages 174–177. 2009.

[46] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzen, Alberto Griggio, and Roberto Sebastiani. Delayed theory combination vs. nelson-oppen for satisfiability modulo theories: a comparative analysis. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):63–99, 2009.

[47] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[48] Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. *ACM Trans. Comput. Logic*, 3(4):604–627, 2002.

[49] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439, 1990.

[50] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[51] Cristian Cadar, Paul Twohey, Vijay Ganesh, and Dawson Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, 2006.

[52] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, pages 363–374, 2009.

[53] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS*, pages 19–33, 2007.

[54] Wei Chen. A formal approach to program inversion. In *CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation*, pages 398–403. ACM, 1990.

[55] Alonzo Church. Logic, arithmetic, and automata. In *Proc. Int. Congr. Math*, pages 23–35. Inst. Mittag-Leffler, Djursholm, Sweden, 1963.

[56] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[57] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

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

[59] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[60] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *LICS*, pages 353–362, 1989.

[61] Michael Colón. Schema-guided synthesis of imperative programs by constraint solving. In *LOPSTR*, pages 166–181, 2004.

[62] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.

[63] Michael Colón and Henny Sipma. Practical methods for proving program termination. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 442–454, London, UK, 2002. Springer-Verlag.

[64] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 302–314, New York, NY, USA, 2009. ACM.

[65] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP*, pages 520–535, 2007.

[66] R L Constable. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[67] Byron Cook. Automatically proving program termination. In *CAV*, page 1, 2007.

[68] Byron Cook, Ashutosh Gupta, Stephen Magill, Andrey Rybalchenko, Jirí Simsa, Satnam Singh, and Viktor Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD*, pages 205–212, 2009.

[69] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.

[70] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM.

[71] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[72] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[73] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.

[74] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *Proc. of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, Scotland, April 2005. Springer.

[75] Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, pages 1–24, 2005.

[76] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[77] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.

[78] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit.* Springer Publishing Company, Incorporated, 2008.

[79] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration.* MIT Press, Cambridge, MA, USA, 1993.

[80] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 51, Washington, DC, USA, 2001. IEEE Computer Society.

[81] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD*, pages 19–32, 2002.

[82] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *CAV '99: Proceedings of the 11th International Conference*

on Computer Aided Verification, pages 160–171, London, UK, 1999. Springer-Verlag.

[83] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[84] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[85] Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for smt solvers. In *CADE-21*, pages 183–198, 2007.

[86] Leonardo de Moura and Nikolaj Bjørner. Z3, 2008. `http://research.microsoft.com/projects/Z3/`.

[87] Leonardo Mendonça de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52, 2009.

[88] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[89] Edsger W. Dijkstra. Program inversion. In *Program Construction,* `http://www.cs.utexas.edu/~EWD/ewd06xx/EWD671.PDF`, pages 54–57, London, UK, 1979. Springer-Verlag.

[90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in CS. Springer-Verlag, 1990.

[91] Edsger Wybe Dijkstra. A constructive approach to the program of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, Sep 1968.

[92] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, 1976.

[93] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 270–280, New York, NY, USA, 2008. ACM.

[94] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag.

[95] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010.

[96] Joe W. Duran. Heuristics for program synthesis using loop invariants. In *ACM '78: Proceedings of the 1978 annual conference*, pages 891–900, New York, NY, USA, 1978. ACM.

[97] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, SRI, 2006.

[98] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, Cambridge, MA, USA, 1999.

[99] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proceedings of SAT 2004*, pages 502–518. Springer Verlag, 2004.

[100] Thomas Emerson and Mark H. Burstein. Development of a constraint-based airlift scheduler by program synthesis from formal specifications. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, page 267, Washington, DC, USA, 1999. IEEE Computer Society.

[101] David Eppstein. A heuristic approach to program inversion. In *IJCAI'85: Proceedings of the 9th international joint conference on Artificial intelligence*, pages 219–221. Morgan Kaufmann Publishers Inc., 1985.

[102] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[103] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[104] A. E. Eichenberger et. al. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, 45(1), 2006.

[105] J. Farkas. Uber die theorie der einfachen ungleichungen. *Journal fur die Reine und Angewandte Mathematik*, 124:1–27, 1902.

[106] Xinyu Feng. Local rely-guarantee reasoning. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–327, New York, NY, USA, 2009. ACM.

[107] Jean-Christophe Filliâtre. Using smt solvers for deductive verification of c and java programs. In *SMT'08.*

[108] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *Computer Aided Verification*, Lecture Notes in Computer Science, chapter 21, pages 173–177. 2007.

[109] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[110] Bernd Fischer and Johann Schumann. Autobayes: a system for generating data analysis programs from statistical models. *J. Funct. Program.*, 13(3):483–508, 2003.

[111] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[112] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202, New York, NY, USA, 2002. ACM.

[113] P. Flener, L. Popelinsky, and O. Stepankova. Ilp and automatic programming: towards three approaches. In *Proc. of ILP-94*, pages 351–364, 1994.

[114] Pierre Flener, Kung-Kiu Lau, Mario Ornaghi, and Julian Richardson. An abstract formalization of correct schemas for program synthesis. *J. Symb. Comput.*, 30(1):93–127, 2000.

[115] Pierre Flener and Serap Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *J. Log. Program.*, 41(2-3):141–195, 1999.

[116] Tim Freeman and Frank Pfenning. Refinement types for ml. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA, 1991. ACM.

[117] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll(t): Fast decision procedures. In *CAV*, pages 175–188, 2004.

[118] Roberto Giacobazzi and Francesco Ranzato. Optimal domains for disjunctive abstract interpretation. *Sci. Comput. Program.*, 32(1-3):177–210, 1998.

[119] Robert Glück and Masahiko Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundam. Inf.*, 66(4):367–395, 2005.

[120] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[121] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[122] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[123] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 89–134. Elsevier, 2008.

[124] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In Kwangkeun Yi, editor, *13th International Static Analysis Symposium, SAS'06*, LNCS 4134. LNCS 4134, Springer Verlag, August 2006.

[125] Denis Gopan and Thomas W. Reps. Lookahead widening. In *CAV*, pages 452–466, 2006.

[126] Denis Gopan and Thomas W. Reps. Guided static analysis. In *SAS*, pages 349–365, 2007.

[127] Erich Grädel, Martin Otto, and Eric Rosen. Undecidability results on two-variable logics. In *STACS '97: Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, pages 249–260, London, UK, 1997. Springer-Verlag.

[128] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, pages 72–83, 1997.

[129] Cordell Green. Application of theorem proving to problem solving. In *IJCAI'69: Proceedings of the 1st international joint conference on Artificial intelligence*, pages 219–239, 1969.

[130] David Gries. *The Science of Programming.* Springer-Verlag New York, Inc., 1987.

[131] A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean programs with an application to C. In T. Ball and R. B. Jones, editors, *18th Conference on Computer Aided Verification (CAV)*, volume 4144/2006 of *LNCS*, pages 358–371, August 2006.

[132] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically refining abstract interpretations. *TR-07-23*, (TR-07-23), 2007.

[133] Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, pages 474–488, 2006.

[134] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In R. De Nicola, editor, *European Symp. on Programming, ESOP 2007*, volume 4421 of *LNCS*, pages 253–267, 2007.

[135] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 375–385, New York, NY, USA, 2009. ACM.

[136] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Component based synthesis applied to bitvector circuits. Technical Report MSR-TR-2010-12, Microsoft Research, 2010.

[137] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.

[138] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 127–139, New York, NY, USA, 2009. ACM.

[139] Sumit Gulwani and George C. Necula. Discovering affine equalities using random interpretation. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 74–84, New York, NY, USA, 2003. ACM.

[140] Sumit Gulwani and George C. Necula. Global value numbering using random interpretation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 342–352, New York, NY, USA, 2004. ACM.

[141] Sumit Gulwani and George C. Necula. Precise interprocedural analysis using random interpretation. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 324–337, New York, NY, USA, 2005. ACM.

[142] Ashutosh Gupta, Tom Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *POPL*, 2008.

[143] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 262–276, Berlin, Heidelberg, 2009. Springer-Verlag.

[144] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 634–640, Berlin, Heidelberg, 2009. Springer-Verlag.

[145] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.

[146] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, New York, NY, USA, 1972. ACM.

[147] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–244, New York, NY, USA, 2004. ACM.

[148] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM.

[149] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[150] Ying Hu, Clark Barrett, and Benjamin Goldberg. Theory and algorithms for the generation and validation of speculative loop optimizations. In *Proceedings of the* 2^{nd} *IEEE International Conference on Software Engineering and Formal Methods (SEFM '04)*, pages 281–289. IEEE Computer Society, September 2004. Beijing, China.

[151] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *CSL*, pages 160–174, 2004.

[152] Susmit Jha, Sumit Gulwani, Sanjit Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *32nd International Conference on Software Engineering*, 2010.

[153] Susmit Jha, Sumit Gulwani, Sanjit Seshia, and Ashish Tiwari. Synthesizing switching logic for safety and dwell-time requirements. In *1st International Conference on Cyber-physical Systems*, 2010.

[154] Ranjit Jhala and Ken McMillan. Array abstractions from proofs. In *CAV*, 2007.

[155] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *Journal of Computer and System Sciences (JCSS)*, –, 2009.

[156] Barbara Jobstmann and Roderick Bloem. Optimizations for ltl synthesis. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 117–124. IEEE Computer Society, 2006.

[157] Barbara Jobstmann, Stefan Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. pages 258–262. 2007.

[158] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[159] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[160] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *Deduction and Applications*, 2005.

[161] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. Type-based data structure verification. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 304–315, New York, NY, USA, 2009. ACM.

[162] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *IR '95: Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, pages 13–22, New York, NY, USA, 1995. ACM.

[163] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[164] Gary A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206, 1973.

[165] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[166] Emanuel Kitzelmann and Ute Schmid. An explanation based generalization approach to inductive synthesis of functional programs. In Emanuel Kitzelmann, Roland Olsson, and Ute Schmid, editors, *ICML-2005 Workshop on Approaches and Applications of Inductive Programming*, pages 15–27, 2005.

[167] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *J. Mach. Learn. Res.*, 7:429–454, 2006.

[168] Kenneth W. Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *ESOP*, pages 505–519, 2007.

[169] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View.* Springer Publishing Company, Incorporated, 2008.

[170] Ramayya Kumar, Christian Blumenröhr, Dirk Eisenbiegler, and Detlef Schmid. Formal synthesis in circuit design - a classification and survey. In *FMCAD '96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 294–309, London, UK, 1996. Springer-Verlag.

[171] Viktor Kuncak, Mikael Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, 2010.

[172] Shuvendu Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–182, 2008.

[173] Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. Predicate abstraction via symbolic decision procedures. *Logical Methods in Computer Science*, 3(2), 2007.

[174] Shuvendu K. Lahiri and Randal E. Bryant. Constructing quantified invariants via predicate abstraction. *Verification, Model Checking, and Abstract Interpretation*, pages 331–353, 2004.

[175] Shuvendu K. Lahiri and Randal E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.

[176] Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Logic*, 9(1):4, 2007.

[177] Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In *CAV*, pages 141–153, 2003.

[178] Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 115–126, New York, NY, USA, 2006. ACM.

[179] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and precise detection of concurrency errors in systems code using smt solvers. In *CAV*, pages 509–524, 2009.

[180] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[181] Jean B. Lasserre. A discrete farkas lemma. *Discrete Optimization*, 1(1):67–75, 2004.

[182] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR '10: Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, 2010.

[183] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, pages 312–327, 2010.

[184] Rustan Leino and Francesco Logozzo. Using widenings to infer loop invariants inside an smt solver. In *WING*, 2007.

[185] Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. 5(2), 2009.

[186] Leonid Libkin. *Elements Of Finite Model Theory (Texts in Theoretical Computer Science. An Eatcs Series)*. SpringerVerlag, 2004.

[187] H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[188] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 553–556, New York, NY, USA, 2007. ACM.

[189] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, New York, '74.

[190] Z. Manna and R. Waldinger. Synthesis: Dreams $\Longrightarrow$ programs. *IEEE Trans. Softw. Eng.*, 5(4):294–328, 1979.

[191] Zohar Manna. *Mathematical Theory of Computation*. Dover Publications, Incorporated, 2003.

[192] Zohar Manna and John McCarthy. Properties of programs and partial function logic. *Machine Intelligence*, 5, 1970.

[193] Zohar Manna and Amir Pnueli. Formalization of properties of functional programs. *Journal of the ACM*, 17(3):555–569, 1970.

[194] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[195] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[196] Maria-Cristina Marinescu and Martin Rinard. High-level specification and efficient implementation of pipelined circuits. In *ASP-DAC '01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 655–661, New York, NY, USA, 2001. ACM.

[197] Mikael Mayer, Philippe Suter, Ruzica Piskac, and Viktor Kuncak. Comfusy: Complete functional synthesis (tool presentation). In *CAV*, 2010.

[198] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposia in Applied Mathematicas*, pages 33–41. American Mathematical Society, 1967.

[199] James McDonald and John Anton. SPECWARE - producing software correct by construction. Technical Report KES.U.01.3., 2001.

[200] K. L. Mcmillan. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 1–13. Springer, 2003.

[201] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.

[202] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[203] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM.

[204] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, pages 330–341, 2004.

[205] Markus Müller-Olm, Helmut Seidl, and Bernhard Steffen. Interprocedural analysis (almost) for free. In *Technical Report 790, Fachbereich Informatik, Universitt Dortmund*, 2004.

[206] Markus Müller-Olm, Helmut Seidl, and Bernhard Steffen. Interprocedural herbrand equalities. In *ESOP*, pages 31–45, 2005.

[207] George C. Necula. Proof-carrying code. In *Conference Record of POPL˜'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[208] Greg Nelson. Verifying reachability invariants of linked structures. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–47, New York, NY, USA, 1983. ACM.

[209] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[210] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[211] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *Journal of the ACM*, 53(6):937–977, 2006.

[212] Gordon S. Novak, Jr. Software reuse by specialization of generic procedures through views. *IEEE Trans. Softw. Eng.*, 23(7):401–417, 1997.

[213] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[214] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[215] A. Pneuli and R. Rosner. Distributed reactive systems are hard to synthesize. In *SFCS '90: Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 746–757 vol.2, Washington, DC, USA, 1990. IEEE Computer Society.

[216] A. Pnueli. In transition from global to modular temporal reasoning about programs. pages 123–144, 1985.

[217] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 1989. ACM.

[218] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 652–671, London, UK, 1989. Springer-Verlag.

[219] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[220] Andreas Podelski and Thomas Wies. Boolean heaps. In *SAS*, 2005.

[221] Franois Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[222] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005.

[223] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *LU Decomposition and Its Applications*, chapter 2.3, pages 34–42. Cambridge University Press, New York, NY, USA, 1993.

[224] Zvonimir Rakamaric, Roberto Bruttomesso, Alan J. Hu, and Alessandro Cimatti. Verifying heap-manipulating programs in an smt framework. In *ATVA*, pages 237–252, 2007.

[225] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.

[226] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[227] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.

[228] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[229] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 159–169, New York, NY, USA, 2008. ACM.

[230] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM.

[231] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996.

[232] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *SAS*, pages 3–17, 2006.

[233] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, pages 318–329, 2004.

[234] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Constraint-based linear-relations analysis. In *SAS*, pages 53–68, 2004.

[235] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, pages 25–41, 2005.

[236] Ute Schmid and Fritz Wysotzki. Induction of recursive program schemes. In *ECML '98: Proceedings of the 10th European Conference on Machine Learning*, pages 214–225, London, UK, 1998. Springer-Verlag.

[237] A. Schrijver. *Theory of Linear and Integer Programming.* 1986.

[238] Helmut Seidl, Andrea Flexeder, and Michael Petter. Interprocedurally analysing linear inequality relations. In *ESOP*, pages 284–299, 2007.

[239] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[240] Nikhil Sethi and Clark Barrett. CASCADE: C assertion checker and deductive engine. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the* 18^{th} *International Conference on Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 166–169. Springer-Verlag, August 2006. Seattle, Washington.

[241] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[242] Richard Sharp. *Higher-Level Hardware Synthesis*, volume 2963 of *Lecture Notes in Computer Science*. Springer, 2004.

[243] D. R. Smith. Kids: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16(9):1024–1043, 1990.

[244] Douglas R. Smith. Designware: software development by refinement. pages 3–21, 2001.

[245] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 167–178, New York, NY, USA, 2007. ACM.

[246] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 136–148, New York, NY, USA, 2008. ACM.

[247] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Prog. by sketching for bit-stream. prgs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–294, New York, NY, USA, 2005. ACM.

[248] Khronos Group Std. The OpenCL specification, version 1.0, online. `http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf`, 2009.

[249] Mark E. Stickel, Richard J. Waldinger, Michael R. Lowry, Thomas Pressburger, and Ian Underwood. Deductive composition of astronomical software from subroutine libraries. In *CADE-12: Proceedings of the 12th International Conference on Automated Deduction*, pages 341–355, London, UK, 1994. Springer-Verlag.

[250] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 29, Washington, DC, USA, 2001. IEEE Computer Society.

[251] Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, 1977.

[252] Tachio Terauchi. Dependent types from counterexamples. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–130, New York, NY, USA, 2010. ACM.

[253] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[254] Wolfgang Thomas. Church's problem and a tour through automata theory. In *Pillars of Computer Science*, pages 635–655, 2008.

[255] Nikolai Tillmann and Jonathan de Halleux. Pexwhite box test generation for .net. In *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, chapter 10, pages 134–153. Springer Berlin Heidelberg, 2008.

[256] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[257] Moshe Y. Vardi. From verification to synthesis. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE*, volume 5295 of *Lecture Notes in Computer Science*, page 2. Springer, 2008.

[258] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 327–338, New York, NY, USA, 2010. ACM.

[259] Martin T. Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, pages 125–135, 2008.

[260] Martin T. Vechev, Eran Yahav, and David F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI*, pages 341–353, 2006.

[261] Martin T. Vechev, Eran Yahav, David F. Bacon, and Noam Rinetzky. Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI*, pages 456–467, 2007.

[262] Richard J. Waldinger. Whatever happened to deductive question answering? In *LPAR*, pages 15–16, 2007.

[263] Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. In *IJCAI*, pages 241–252, 1969.

[264] Chao Wang, Zijiang Yang, Aarti Gupta, and Franjo Ivancic. Using counterex. for improv. the prec. of reachability comput. with polyhedra. In *CAV*, pages 352–365, 2007.

[265] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[266] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

[267] Nicholas Wirth. *Systematic Programming: An Introduction.* Prentice Hall PTR, 1973.

[268] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM.

[269] Yichen Xie and Alexander Aiken. Saturn: A sat-based tool for bug detection. In *CAV*, pages 139–143, 2005.

[270] Daniel M. Yellin. *Attribute grammar inversion and source-to-source translation.* Springer-Verlag New York, Inc., 1988.

[271] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–234, New York, NY, USA, 2008. ACM.

[272] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(5):337–343, 1977.