TEL AVIV UNIVERSITY אוניברסיטת תל-אביב

The Raymond and Beverly Sackler Faculty of Exact Sciences

The Blavatnik School of Computer Science

Size Reduction for Program Analysis

Thesis submitted for the degree of Doctor of Philosophy

by

Oren Ish Shalom

This work was carried out under the supervision of

Professor Noam Rinetzky

Submitted to the Senate of Tel Aviv University

March 2021

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Noam Rinetzky. This whole thesis would not have been possible without him. Literally. His original ideas, guidance and persistence helped in every aspect of this Phd. Noam, thanks.

To Shachar Itzhaki, for fruitful and years-long collaboration. For interesting discussions about SMT solvers, syntax guided synthesis, loop invariants and everything related to programming languages really. To Sharon Shoam, the terminator of theorem proofs, who taught me how to see the mathematical beauty of being accurate. Last, but certainly not least, I want to thank my family, for making it all happen.

Abstract

This thesis explores the notion of a program “size” in various settings. It demonstrates that once a size of a program is properly defined, reducing that size may sometimes induce a “smaller” equivalent program, that is easier to analyze.

For instance, inverting a string manipulating procedure might be an easier task, when its underlying alphabet is *not* the entire set of characters, but rather a smaller, “representative” set. Another example involves the equivalence of hand written string loops to string library functions. Proving such equivalence for loops encoutered in the wild is tricky. However, we managed to observe that somewhat often these handwritten loops have an interesting size-agnostic property, which justifies equivalence checks *only on strings of constant length*. The bounded checks then become feasible.

We generalized the reduction principal, and showed that it can be *repeatedly* applied to formulate a novel kind of “induction on space” to prove program specifications. Our new induction is in a sense orthogonal to the standard “induction on time” applied via loop invariants in many frameworks. Evidently, our approach was able to automatically prove program specifications that were out-of-reach for existing tools.

Last, we extended the concept of repeated size reduction, so that it can support *several* reductions rather than just one. This broader viewing angle allowed us to reason about computational complexity of programs that failed automatic complexity classification before.

תמצית

התזה חוקרת את המושג "גודל תוכנית" בהקשרים שונים. היא מדגימה שבהינתן גודל שכזה, יצירת תוכנית שקולה, שגודלה קטן יותר, עשויה להיות קלה יותר לניתוח. למשל, מציאת הופכית לפרוצדורה שפועלת מעל מחרוזות, ייתכן שניתוחה ייעשה קל משמעותית אם במקום להשתמש בכל התוים הקיימים נזהה קבוצה מייצגת, קטנה, של תוים, ונשתמש בה. כדוגמא נוספת, נתבונן בהוכחת שקילות בין לולאות שפועלות על מחרוזות, לפונקציות של הספריה הסטנדרטית. הוכחת שקילות כזאת עלולה להיות קשה למדי באופן כללי, וכששמנו לב שאחוז גבוה למדי מאותן לולאות מקיים אי-תלות בריאה בגודל המחרוזת, ניצלנו זאת להוכחה שמספיק להוכיח שקילות רק עבור מחרוזות שגודלן חסום. החסם על גודל המחרוזת מאפשר הוכחה מיידית של שקילות. המשכנו והכללנו את רעיון רדוקציית הגודל התוכנית, כך שניתן יהיה להפעיל את הרדוקציה הפעלות חוזרות. בכך למעשה הצגנו סוג חדש ומקורי של אינדוקציה - אינדוקציה על מקום - להוכיח ספסיפיקציה של תוכניות. האינדוקציה החדשה שהצגנו, היא למעשה אורתוגונלית לאינדוקציה הנפוצה שבה משתמשים להוכחת תוכניות בטכניקה של אינבריינטות בלולאות. ואכן, הכלי שפיתחנו היה מסוגל להוכיח ספסיפיקציות של תוכניות שהיו בלתי אפשריות להוכחה בשיטות קיימות. בסופו של דבר, הצלחנו לשלב כמה רדוקציות גודל באותו המנגנון, והרחבה זאת אפשרה לנו להוכיח סיבוכיות זמן ריצה של קוד אימפרטיבי באופן אוטומטי לגמרי. התוכניות שאת זמן הריצה שלהן הצלחנו לנתח לא היו ניתנות לניתוח בשיטות הקיימות לפנינו.

תקציר

ניתוח קוד של תוכנית ובה מס לא סופי של מצבים היא בעיה עתיקה בהנדסת תוכנה. המוטיביציה בצמצום מספר המצבים בתוכנית כדי להקל על בדיקתה היא אם כן אינטואיטיבית וברורה. באופן ספיציפי, כאשר מספר המצבים בתוכנית סופי, הרי שניתוחה נעשה אפשרי פעמים רבות, ולכן סופיות מספר המצבים הוא בעליל יעד "מתגמל" מבחינת הרדוקציה. דוגמא לכך ניתן לראות בבדיקת מודלים חסומים, אשר יפרשו למשל לולאות על פני מס קבוע של איטרציות, ובכך מצד אחד עלולים לאבד דיוק, אך מצד שני יוכלו "לזכות" בתוכנית שניתוחה יהיה בעל כושר היתכנות גבוה. השאיפה לרדוקציה למספר מצבים סופי מוסברת כאמור על ידי היכולת לבדוק כל מצב, וניצנים של הרעיון המובן הזה קיימים בענפים אחרים במתמטיקה זה מאות בשנים. חישבו על משפט ארבעת הצבעים, שדן בשאלה האם בגרפים מישוריים קיימת צביעה חוקית בארבעה צבעים. כבר לפני מאה שנים ויותר הייתה קיימת המוטיבציה להקטין את מספר המצבים לקבוצה סופית ולבדוק רק אותה.
אילו שאלות ניתן לשאול בהקשר לרדוקציה?

- האם הרדוקציה שלמה?
- האם המערכת המוקטנת קלה יותר לניתוח?
- האם ניתן לבצע הפעלות חוזרות של אותה הרדוקציה? ובכך להקטין עוד ועוד את מרחב המצבים?
- האם ניתן להשתמש בשיטה כאבן בניין במסגרות אחרות?
- האם ניתן לשלב כמה סוגים של רדוקציה באותה המערכת?

התזה מנסה לענות על שאלות אלה ואחרות באמצעות סדרה של ניתוח בעיות בשפות תכנות באמצעים של רדוקציית גודל.

לרוב, הרדוקציות המוצגות בתזה אינן למרחב מצבים סופי במובן המתמטי, כי אם למרחב שניתן לבדיקה. נתבונן בקוד המצורף. מס המצבים בו הוא אמנם אינו סופי, אך באמצעות ניתוח סימבולי, אפשר בקלות לבדוק שהקוד מקיים את הדרישה הכתובה. למעשה, מה שכן סופי כאן הוא מספר המסלולים, ומכאן קלות הבדיקה.

```
void foo(unsigned int x, unsigned int y) {
    if (y > 0) {
        assert((x/y)*y <= x);
    }}
```

בהמשך לאותו נושא נציין את קלות הלמידה של מודל חישובי. בלמידה מפוקחת קלאסית, מרחב החיפוש וקבוצת האימון יכולים להשתפר במידה רבה כאשר יודעים לגזום מצבים יתירים. חקרנו את הנושא הזה בהקשר של למידה של פרוצדורות שמסננות מחרוזות לא חוקיות, והפיכתן. גילינו באופן מרתק שצמצום האלף בית ממאות תוים למס בודד של תוים השיג שיפור דרמטי בתוצאות הלמידה. לעיתים, הפעלת הרדוקציה פעם אחת אינה מספיקה. שכן סופיות, ואפילו ניתנות לניתוח סימבולי מיידי הן דרישות חזקות מדי. כשהבנו זאת פיתחנו מנגנון משוכלל של הפעלות חוזרות ונישנות של אותה הרדוקציה, והגדרת עיקרון אינדוקטיבי חדש. למשל, בסכימה של מערך מספרים משני הכיוונים יתקבל אותו הסכום. הצרנה והוכחה אוטומטית של קוד פשוט למדי, עם תכונה מתבקשת זאת היה בלתי ניתן להוכחה אוטומטית בשיטות קודמות.

```c
void sum_bidi(int a[], int n) {
    int l = 0, r = 0;
    for (int i = 0; i < n; i++) {
        l += a[i];
        r += a[n - i - 1];
    }
    assert(l == r);
}
```

אנחנו ניגשנו לבעיה מכיוון שונה והגדרנו "גודל" לריצת התוכנית - גודל המערך. ואז בדקנו האם מהרצת אותה התוכנית בדיוק על מערך שגודלו קטן בתא אחד בלבד, ובו מתקיימת התכונה נובע שהיא מתקיימת גם במערך המקורי. כך הפעלנו את הרדוקציה קונספטואלית עוד ועוד פעמים עד להגעה למקרה הבסיס, מערך שגודלו 2, ושם ניתן לבדוק את התכונה עם ניתוח סימבולי.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Analysis of infinite state software systems is most probably as old as software developement itself. The motivation of reducing the number of states in a system to facilitate its analysis is intuitive and clear. In particular, checking a *finite* subset of system states is often feasible, and therefore embodies an obvious size-reduction goal. Consider for example the case of analyzing C programs with the C bounded model checker (CBMC) [91]. The bounded model checker will unwind / unfold all loops in the program to perform at most k iterations, thus reducing the infinite number of states to be finite. Sure enough, this paradigm is not complete, but in practice achieved great results. Once the number of states is finite, a brute force check of them can often yield beneficial feedback. The aspiration to "go-finite" is exhibited also in other related fields. Think of the celebrated four colors theorem [119], where planar graphs are reduced to a finite subset which is then checked individually. Indeed, this simple concept is around one hundred years old. Evidently, the success or failure of this method depends on the size-reduction scheme.

- Is the reduction complete?
- Can the reduced system be analyzed effectively?
- Can the reduction be used repeatedly?
- Is the approach compositional?
- Can multiple reductions be applied?

This thesis tries to explore these questions and others in a sequence of works tackling program analysis problems.

One time size reduction For the most part, the reduced instances in our works are not *mathematically* finite, but rather lend themselves to effective analysis. One technique that demonstrates that a finite set of states is *not* always mandatory is symbolic execution [89]. The code in Figure 1.1 contains an "infinite" (oh well, 2^{128}) number of

```
void foo(unsigned int x, unsigned int y) {
    if (y > 0) {
        assert((x/y)*y <= x);
    }}
```

Figure 1.1: the code has "infinite" states, but is easily checked with symbolic execution

states, but can be easily checked as the following formula sent to the underlying SMT solver:

```
        (=> (not (= y 0)) (<= (bvmul (bvdiv x y) y) x))
```

This is because the number of *paths* is finite rather than the number of states. Generally, we really aim for "analyzability" rather than "finite". In that aspect, we explored how classic supervised machine learning (ML) algorithms might benefit from reducing their search space. Here too, the finite number of states requirement could be relaxed, as long as the search space is "represented" by a proper training set. The context of ML in program analysis is an active area of research that is dominated mostly by modern methods ([121], [72] etc.). It was encouraging to be able to apply some of the classic, older algorithms like ([82] and [106]) with the help of size reduction.

To observe the effect of our method, consider a program that manipulates filenames. It may choose to keep only the containing directory (string up to the last slash character) or extract its suffix (string from the last dot character) or strip `"./"` from the begining of the name etc. Many such string manipulating functions exist in systems code, and modeling them as transducers has many advantages, specifically, the ability to easily invert them. We were able to use abstract interpretation [36] to facilitate the learning process of such sanitizers. Our analysis was able to reduce the alphabet size drastically, which in turn improved both the sampling process, and the actual internals of the algorithms we experimented with. The main observation that enabled it was the fact that most characters in these sanitizers are simply copied "as-is" from the input string to the output string. So, for the learning process, they can be represented by a *single* meta character. Very few characters violate this "goodness" property. For instance, in the context of file names they would probably be slash, backslash, dot and colon.

Repeated size reduction Sometimes, a single size reduction is simply *not* enough. Conceptually, one could re-apply the size reduction again (and again). When the conditions are right, this chain of size reduction applications is a novel form of induction we managed to harness for analyzing complicated code: One of the fundamental problems of computer science, proving with mathematical certainty that a program respects a given specification. Since the works of Floyd [50] and Hoare [71], the academic struggle to automatically infer inductive invaraiants continues. Refer the innocent looking loop

```
void sum_bidi(int a[], int n) {
   int l = 0, r = 0;
   for (int i = 0; i < n; i++) {
      l += a[i];
      r += a[n - i - 1];
   }
   assert(l == r);
}
```

Figure 1.2: computing the sum of elements in an integer array from both directions and verifying the sum is the same. Traditional methods have a hard time proving this specification, since finding a first order induction invariant is tricky. In fact, to the best of our knowledge, one doesn't even exist.

in Figure 1.2 that sums the elements of an integer array from both directions, and then asserts the sum is equal. Automatic verification tools like [145] fail to prove this specification. Actually, to the best of our knowledge, no first order inductive invariant exists that can aid in proving this spec. Our novel approach for verification first identifies a *size* for the execution state. In this example, it is the size of the array, but it can generally be any explicit or even hidden quantity inside the execution state. Then, we define a state size reduction Υ in a way that given a trace $\tau = \{\sigma_i\}$, applying Υ on τ will "induce" a corresponding trace τ' with "smaller" size with the following exciting proprty (where ϕ is the desired correctness property):

$$(\tau' \models \phi) \Rightarrow (\tau \models \phi)$$

Once this is established, all that remains is a (conceptual) re-application of the state size reduction until we "hit" a minimal unreducable size. This forms the basis of our induction, and needs to be checked independently, typically with symbolic execution. Sounds almost too easy, right? the trick is how to make sure that:

$$\textit{state to state size reduction} \Rightarrow \textit{trace to trace size reduction}$$

Put in other words, how to formulate local, state-level, checkable (by an SMT solver) properties so that they induce global, trace-level properties? This is thoroughly explained in chapter 4. Attentive readers might be concerned at this point regarding the needed ingredients for our framework. In a sense, they are right. The framework needs to come up with a definition for the state size, and the state-to-state size reducer (which we call *state squeezer*, or just *squeezer*). The mechanism for an efficient generate-and-test workflow is described in detail in chapter 4.

Multiple repeated size reductions Encouraged by the experimental results of our state squeezers, we set out to tackle another notorious problem: automatic resource management. The motivation is almost self explanatory: can the ability to perform whole trace size reduction, be used to infer a compexity recurrence formula? This called for two crucial improvements over the original squeezers paradigm. First, the correspondance between a trace and its squeezed counterpart must be tight. That is, their length difference must be carefully bounded. Second, and even more importantly, we found that a repeated application of a single squeezer was not enough. It yielded only linear recurrences that were easily solved by existing tools. Our key result was achieved when we started considering *multiple* state squeezers, "breaking" the original trace into pieces, then reducing each piece individually. By doing so, we were able to automatically deduce complexity upper bounds that were beyond the reach for existing methods.

1.1 Main results

1.1.1 Alphabet size reduction for more efficient machine learning

Backward analysis methods usually start with a user specified code location, examining the constraints on the data flow to reach it. Ideas like [65] deomonstrate that "backward" symbolic execution can be far more efficient in some contexts, exploring only "relevant" paths and not the entire code. In accordance to this concept, we set out to tackle the learning process of string to string procedure (pseudo) inversion. Indeed, viewed as mathemtical entities, procedures do not always have inverses. Formally speaking, they are not even mathematical functions, as they may not terminate on some of their inputs. Consequently, our reasearch question was limited to the following: Given a string to string procedure P, and a concrete output string s_{out}, find an input string s_{in} that satisfies:

$$P(s_{in}) = s_{out}$$

The motivation behind the focus of string to string procedures lies in the abundance of such procedures in system code. Typically, they "santize" their input in some way, and then usually pass in to a "home-made" *getopt* or its library equivalent. We observed that quite frequently, the sanitation code does a repetitive task *at the character level*, completely agnostic to its location in the input string. For instance, replacing slashes to back slashes. Or escaping (unescaping) certain characters etc. Like the code in Figure 1.3. We searched for ML algorithms that suit this description, and found out

```
void escapePath(in,out) char *in,out; {
    while (1) {
        char c = *in++;
        if (c == 0) return;
        else if (c == '(' || c == ')' || c == '\\')
        {
            *out++ = '\\';
        }
        *out++ = c;
    }}
```

Figure 1.3: string sanitizing procedure taken from gcc. Sanitizers like this one are abundant in system code. most of them act on the "character level", completely agnostic to their location inside the string.

two algorithms [106] and [82]. The former comes from a biological background, referring to insertion / deletion / replacement of DNA's "letters". The latter uses transducers, which resemble finite automata, only with the ability to print characters as edges are traversed. Both algorithms support the concept of inversion. That is, once a model was learned for P, the corresponding model for P^{-1} is easily deduced. Furthermore, P^{-1} can be learned directly, which is usually better, as described in Chapter 2.

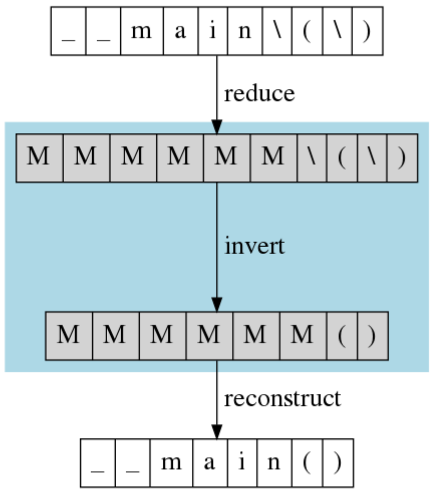


Figure 1.4: bird's view of the alphabet reduction for ML. Abstract interpretation determines the set of good characters. Then, a single meta character (M) is used to represent all of them. The training set for the model uses a reduced alphabet: $BAD \cup \{M\}$. The output string for inversion encodes its good characters, and applies the learned inverse. Then, the resulting string is reconstructed to use the original alphabet. Reconstruction is possible thanks to the goodness property.

Our contribution Unfortunately, the aforementioned algorithms "suffer" from the huge search space. It manifests itself in both the sampling process, and the actual internals of the algorithms. For example, OSTIA [82], which is based on state merging will demonstrate a poor quality of merging when using the original alphabet of all possible characters. Enter abstract interpretation [36]. We formalized a dedicated abstract domain that will distinguish "good" characers from "bad" ones. Good characters, which often form the vast majority, are simply copied "as-is" from the input to the output. Bad characters may sometimes be omitted, inserted or replaced. The property if formally defined as:

$$\boxed{a \text{ is a good character} \Leftrightarrow \forall s_1 \in \Sigma^*.\forall s_2 \in \Sigma^*.P(s_1 \cdot a \cdot s_2) = P(s_1) \cdot a \cdot P(s_2)}$$

Then, *we were able to use a single meta character to represent all good characters.* Refer to Figure 1.4 for a demonstration of our approach.

1.1.2 String length reduction for program equivalence

Verifying the equivalence programs problem is undecideable. Still, the motivation to establish equivalence is paramount, as it can help in many aspects of software developement: testing, refactoring, licensing and much more. Consequently, we limited our research question to proving equivalence of hand written string loops to string library functions.

Our contribution We were able prove that once the underlying program satisfies a set of static and dynamic checks, the equivalence problem is soundly reduced to a *bounded* equivalence problem, checkable with symbolic execution. Consider Figure 1.5,

```
char *strrchr_tag(const char *f)
    char *last = f + strlen(f);
    for (char *p = f; *p; p++) {
        // if (p == f) continue;
        if (*p == '/') last = p;
    }
    return last;
}
```

Figure 1.5: hand written string loop equivalent to `strrchr(f,'/')`. Our approach can detect that by soundly reducing the equivalence problem to a bounded instance handling only "short" strings.

that shows a code operationally equivalent to `strrchr(f,'/')`.

Once the comments is removed, the resulting code is clearly distinguished from `strrchr(f,'/')`. What would be the "best" counterexample proving they are distinct? In essence, this formulates our goal: finding counterexamples bounded by a "small" size. Refer to Figure 1.6 that illustrates how the discriminating string `"/sbin"` is size-reduced to a smaller counterexample `"/n"`. The size reduction kept the first slash character, as it is responsible for the difference. The last character is copied to handle backward scans similarly. For more details see Chapter 3.

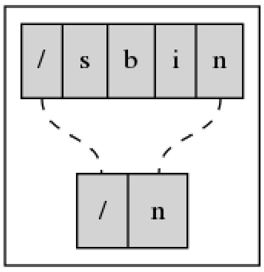


Figure 1.6: illustrating the conceptual size reduction of the counterexample. string `"/sbin"` proves that Figure 1.5 without the comment is *not* `strrchr(f,'/')`. The middle characters can be soundly discarded, since the loop is guaranteed to be "memoryless". This results in a bounded length string witness.

Hand written string loops are messy. Programmers use many styles to write such loops. They scan strings forward, and backward use pointer comparisons, or comparing indices etc. To accomodate for this diversity, our sytactic constraints were very limited: single non nested loop, operating on a single constant string. The dynamic constraints however were far more limiting, essentially enforcing a property we called “memoryless”. That is, the loop is restricted from carrying information from one iteration to the next. To achieve that, we instrumented to code and fed it to a symbolic execution engine. The instrumented code made sure that reads, writes and comparisons respect our constraints.

1.1.3 State squeezer for verification

Automatic verification of looping programs is a hard problem. Existing methods may use various techniques to infer inductive invariants, but they will all use them to perform *induction on time*. We formulated a novel kind of induction, induction on space, if you will. It assigns a size to the execution state, and then performs an inductive proof of correctness on that size. It can be the length of a string, the size of an array or the sum of two integer variables in the state. Anything really, as long as one can perform induction on it. To achieve that, we needed to establish conditions that can guarantee that our state-to-state reduction will induce a trace-to-trace reduction. Figure 1.7 illustrates this concept. For more details see Chapter 4.

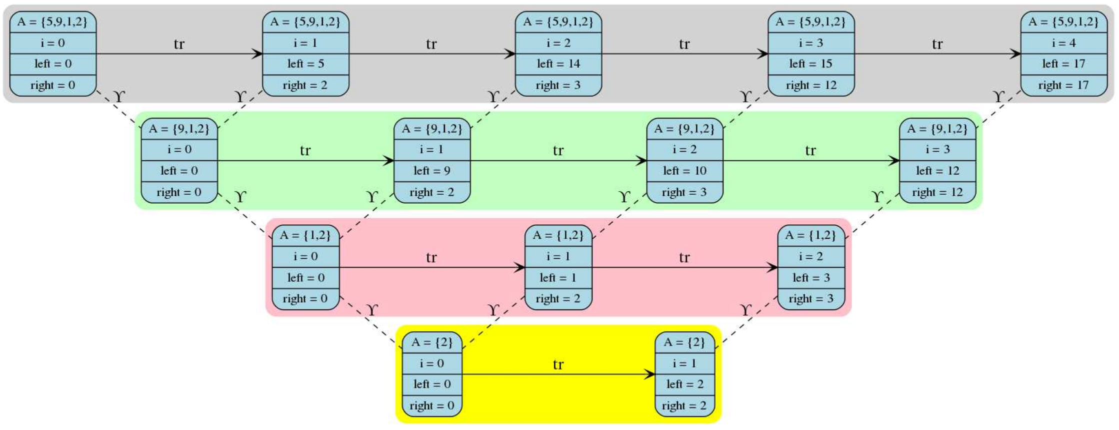


Figure 1.7: repeated application of size reduction. the state-to-state squeezer induces a trace-to-trace size reduction. the correctness of a trace follows from the correctness of the trace below it. The yellow lowermost trace is verified indepndently with symbolic execution. ϒ: `remove(a,0); if (i) { i--; left -= a[0]; right -= a[n-i] }`.

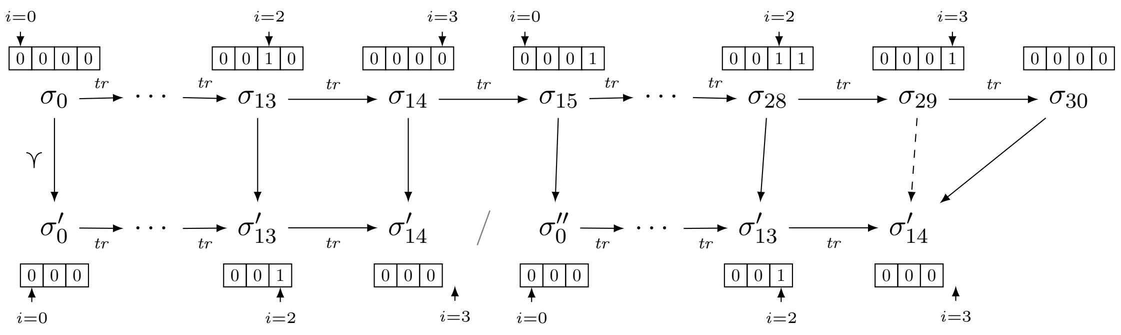


Figure 1.8: string sanitizing procedure taken from gcc

1.1.4 State squeezers for complexity analysis

Determining upper bounds on the time complexity of a program is a fundamental problem with a variety of applications, such as performance debugging, resource certification, and compile-time optimizations. Automated techniques for cost analysis excel at bounding the resource complexity of programs that use integer values and linear arithmetic. Unfortunately, they fall short when execution traces become more involved, especially when data dependencies may affect the termination conditions of loops. In such cases, state-of-the-art analyzers have shown to produce loose bounds, or even no bound at all. We propose a novel technique that generalizes the common notion of recurrence relations based on ranking functions. Existing methods usually unfold one loop iteration, and examine the resulting relations between variables. These relations assist in establishing a recurrence that bounds the number of loop iterations. We propose a different approach, where we derive recurrences by comparing whole traces with whole traces of a lower rank, avoiding the need to analyze the complexity of intermediate states. We offer a set of global properties, defined with respect to whole traces, that facilitate such a comparison, and show that these properties can be checked efficiently using a handful of local conditions. For more details see Chapter 5.

Chapter 2

Alphabet Reduction for Inverse Finding

This chapter is based on the results published in [77].

2.1 Introduction

Recently, there has been a growing interest in applying machine learning techniques to challenging program analysis problems [60, 108, 109, 118, 123, 124, 151]. In this chapter, we address the dual question: Can program analysis techniques help machine learning? We perform a preliminary case study in which machine learning algorithms are used to invert *string manipulating procedures* (SMPs), and show that in this domain the answer is reassuringly positive. Interestingly, the models generated by the machine learning algorithms can themselves be of help to other program analysis tools. Specifically, they can help improve the coverage of symbolic execution tools such as KLEE [27]. Thus, we find ourselves in a pleasant situation where program analysis assists machine learning to help program analysis.

Research problem. Let Σ be a (possibly infinite) set of *characters* ranged over by a meta-character σ. A *string* $s \in \overline{\Sigma}$ is a finite sequence of *characters*. Given a deterministic SMP $p()$ which transforms input strings $s \in \overline{\Sigma}$ to output strings $p(s) \in \overline{\Sigma}$, where $p(s)$ denotes the output $p()$ returns when invoked on s, our goal is to find a *partial right pseudo-inverse* of a (possibly non-injective) p, i.e., a function $p^{-1} : \overline{\Sigma} \hookrightarrow \overline{\Sigma}$ such that

$$\forall s' \in \overline{\Sigma}.\, p^{-1}(s') \neq \bot \implies p(p^{-1}(s')) = s'\,.$$

Clearly, the problem is decidable as we can always have $p^{-1} = \bot$. Another trivial solution is to define p^{-1} to be the identity function wherever it coincides with the inverse of p, i.e., have

$$p^{-1}(s') = \begin{cases} s' & p(s') = s' \\ \bot & \text{otherwise}\,. \end{cases}$$

Thus, the challenge is to come up with a function p^{-1} with non-trivial domain of definition. Ideally, p^{-1} should be able to help automatic test generation, as we discuss now.

Motivation. The ability to invert string-manipulating procedures (SMPs) is useful, for example, in the context of tools for automatic test generation, e.g., KLEE [27]—a state-of-the-art symbolic execution engine. These tools automatically generate test cases, aiming to exercise as much of the program's code as possible. For example, KLEE uses various heuristics to explore the program's code: it continuously selects code paths that lead to not-yet-explored statements, applying a satisfiability-modulo theory solver (SMT) [39, 51] to determine whether a path is feasible, i.e., that there is an input which causes the selected path to be executed. As the exploration is path-sensitive, the tool may inspect an exponential number of code paths when exploring a loop containing a conditional, while generating formulae whose size is proportional to the length the inspected path. As a result, it can be challenging to cover a statement following a call to an SMP p which can be reached only if p returns a specific output s'. Ideally, when the engine reaches such a difficult-to-handle branch condition, one would want the symbolic execution engine to abandon the execution path it followed within $p()$, and instead, try to execute it "backward" to produce s'. Our technique equips the engine with such an ability by generating an "inverse shortcut"—a function that inverts the behavior of $p()$ without the cost of a path-by-path exploration.

Learning pseudo-inverses. Our goal is to help tools such as KLEE to find inputs which drive SMPs to produce desired outputs. We suggest to do it using machine learning: Given an SMP $p()$ mapping input strings s to output strings $p(s)$, we apply a supervised machine learning algorithm to learn a *model* of a pseudo inverse of p. The model should be capable of *translating* strings, i.e., given a string s' the model should be able to find a string s which it predicts to be an inverse of s under p.

Roughly speaking, producing the model entails generating a set of arbitrary inputs $\{s_i\}_{i=1}^n$, executing p on each input, thus producing a training set $T = \{(p(s_i), s_i)\}_{i=1}^n$, and finally training the algorithm on T. Note that T is comprised of pairs of strings mapping the *output* of $p()$ to the input which produced it. Thus, by training the algorithm using T, we in fact learn a model which approximates the behavior of a pseudo-inverse of $p()$.

The challenge. Unfortunately, a naive generation of the training set can be extremely inefficient in the sense that many output/input pairs effectively expose the same behavior. For example, consider an SMP which adds an escape character before tab and newline characters in its input. If we use randomly generated training sets, $p()$ will act as the identity function on most of the examples, and it might require a very large training set to expose other, more "interesting", behaviors: A randomly constructed string with 10 resp. 88 characters has a 92% resp. 50%, chance not to include a tab or a newline character. As a result, the machine learning algorithm might find it difficult to generalize the interesting cases (or outright ignore them, considering them to be noise), and end up learning a bad approximation of the inverse.

Our solution: Learning with reduced alphabets. To remedy the above situation, we propose a static analysis which allows to reduce the alphabet from which the training set examples are drawn, without scarifying the ability to encode any "interesting" behaviors. In fact, our approach increases the chances of generating "interesting" examples by reducing the part of the alphabet from which "non-interesting" examples are drawn. Intuitively, we identify a *set of good characters* (2) whose only effect on the analyzed procedure is to be copied in an order-preserving manner from the input to the output. Our *key insight* is that given such a set, it is possible to expose all the interesting behaviors of a procedure using an alphabet containing a single representative good character, and deduce the effect of an SMP on a string containing characters which were not found in the reduced alphabet from its effect on a *similar* string (1) whose characters do.

Alphabet reduction via static analysis. To automate the selection of good characters, we designed a static analysis that can find the set of good characters for a given string manipulating procedure. Our analysis handles a restricted class of procedures. In this class, a procedure takes a string input and returns a string output. The procedure can read its input from left-to-right, from right-to-left, or in both directions, however each input character is read only once. The procedure is allowed to use variables that can hold character values and employ conditionals and loops, where condition can only test whether a character variable is equal or not to another variable or to a constant. While simple, we found that our restricted programming model is still expressive enough to handle a variety of procedures.

Technically, the static analysis maintains an order between the variables according to the position of the input character that they got their value from. Essentially, whenever a variable x containing an input character is written to the output out of turn, i.e., before any other variable y holding an input value σ which was read off the input x was set, the analysis determines that the σ cannot be good. Similarly, writing a constant character `const` to the output leads the analysis to dictate that `const` is not good either.

Implementation and experimental evaluation. We implemented our analysis in a tool called StrInver. We applied it to invert a small selection of procedures written in C and taken from real-life software. (The tool operates on LLVM bitcode.) We then ran KLEE on a simple program containing a call to the SMP followed by an erroneous command whose execution is predicated on the SMP returning a particular output. Our analysis succeeded to find useful pseudo-inverses of the particular outputs in a few seconds, whereas KLEE, a state-of-the-art symbolic execution tool, failed to find an input which lead to the bug.

Main contributions. The main conecptual contribution of this chapter is the observation that when a machine learning algorithm is used to discover properties of programs, it might be possible to use program analysis to help direct the choice of the training set towards examples that expose interesting behaviors. The main technical contribution of our work is the concretization of this observation by developing a static analysis algorithm which allows to reduce the size of the alphabet from which examples are drawn when learning pseudo inverses of a restricted class of string manipulating

```
char* escapeWS(char *in) {
  char *out = malloc(MAXLEN);
  char *s = out;
  while (*in != 0) {
    if (*in=='5'||*in=='8')
      *s++ = '$';
    *s++ = *in++;
  }
  *s = *in;
  return out }
```

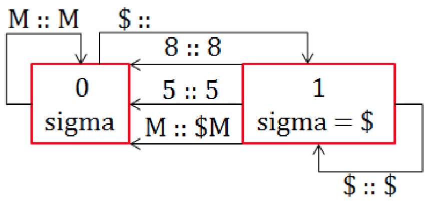


Figure 2.1: A simple SMP[1] and a transducer approximating its pseudo-inverse under the reduced alphabet

procedures. The main practical contribution of our work is the implementation of the analysis and an empirical evaluation where we applied our technique to a small selection of procedures taken from real-life software. Also, to the best of our knowledge, the idea of using machine learning to invert string-manipulating procedures is novel.

2.2 Overview

In this section, we motivate our research problem and give a high-level overview of our approach by walking the reader through a series of examples.

Example 1. *Figure 2.1 shows procedure `escapeWS()`, an SMP which returns a copy of its input string with a $ character before every 5 and 8 character it contains.*[1] *For example, given the input string "`Ali5BaBa8`", `escapeWS()` outputs "`Ali$5BaBa$8`".*

To motivate the need for computing inverses of SMPs, assume that we wish to symbolically execute a program which aborts in an error state only if `escapeWS()` produces a particular output, e.g.,

```
ret = escapeWS(input);
if (strcmp(ret, "Ali$5BaBa$8") == 0) abort();
...
```

Note that `escapeWS()` produces "`Ali$5BaBa$8`" only if it is given "`Ali5BaBa8`" as input. To find this input, symbolic execution engines such as KLEE would have to

[1]The procedure is based on a GCC procedure which adds an escape character before tab and newline characters. For clarity, we replaced the whitespace characters with more visible characters. For simplicity, we removed code concerning array bound checking.

follow a very particular code path, namely the one in which the loop body is executed nine times and the true branches of the first and second `if` statements are taken after reading the fourth and ninth input characters, respectively.

Our goal is to help tools such as KLEE to find inputs which drive SMPs to produce specific desired outputs. We would like to use an off-the-shelf supervised machine learning algorithm and train it to generate a model of the inverse function. While it is quite easy to generate random inputs, most of them will be non-representative of the function's actual semantics, necessitating large training sets, as we noticed in our experiments. Consider again procedure `escapeWS()` shown in Figure 2.1. It is easy to see that the procedure acts rather uniformly on most of the input characters: all the characters are copied from the input string to the output string in an order-preserving fashion, only characters 5 and 8 trigger an insertion of the '\$' character. Thus, if the input string does not contain characters 5 and 8 then the procedure acts as the identity function. Thus, intuitively, all the "interesting" behaviors of the procedure should be detected by considering string comprised of four characters: 5, 8, \$, and an arbitrary character M representing all other characters.

Inverting SMPs with reduced alphabets. To remedy the above situation, we propose a static analysis which allows to reduce the alphabet from which the training set examples are drawn, without scarifying the ability to encode any "interesting" behaviors. Our *key insight* is that if we can identify that the SMP does not distinguish between several characters then it might be possible to expose all the interesting behaviors of a procedure using an alphabet containing a single representative character of the set, and deduce the effect of an SMP on a string containing characters which were not found in the reduced alphabet from its effect on a *similar* string whose characters do. In this paper, we focus on a particular class of indistinguishable characters, those which the procedure act on as, essentially, the identity function, and refer to these characters *good* characters.

Definition 1 (Similar strings). *Let $S \subseteq \Sigma$ be a set of characters. Strings $s_1 \in \overline{\Sigma}$ and $s_2 \in \overline{\Sigma}$ are* similar up to S, *denoted by $s_1 \sim_S s_2$, if $|s_1| = |s_2|$, where $|s|$ denotes the length of a string s, and for every $i = 1..|s_1|$, it holds that either $s_1(i) = s_2(i)$ or $\{s_1(i), s_2(i)\} \subseteq S$.*

Definition 2 (Good characters)**.** *a is a good character if and only if:*

$$\forall s_1 \in \Sigma^*.\forall s_2 \in \Sigma^*.P(s_1 \cdot a \cdot s_2) = P(s_1) \cdot a \cdot P(s_2)$$

Lemma 1. *Let $G \subseteq \Sigma$ be a set of good characters for $p()$. For any two strings s_1 and s_2, if $s_1 \sim_G s_2$ then $p(s_1) \sim_G p(s_2)$.*

Given a procedure $p()$, our static analyzer, discussed next, finds a set of good characters for $p()$ by way of elimination. For example, our analyzer finds that the set $\{\$, 5, 8\}$ is bad for procedure `escapeWS()`. We use this result to construct a *reduced alphabet* $\Gamma = \{\$, 5, 8, M\}$, where $M \in \Sigma$ is the single representative of the good characters, which we refer to as a *metacharacter*. Given Γ, we apply the aforementioned learning process; this time, however, we generate the training set by only drawing examples from Γ. Our static analysis is independent of the machine learning algorithm used to find the inverse. In our experiments, we use two such algorithms: OSTIA [82], which learns a *transducer*, and the other is a non deterministic model for character insertion/replacement/deletion based on the Needleman Wunsch alignment algorithm [106]. (See Section 2.6.) A transducer is a finite state machine that, instead of accepting or rejecting an input string, outputs characters upon transition. Figure 2.1 depicts a transducer approximating the pseudo inverse of `escapeWS()` which OSTIA has learned using a training set comprised of 100 strings, randomly generated over Γ. (We explain the graphical notations in Section 2.6.1.) In our example, the transducer has two states, where state 0 is the initial one. An edge labeled $\overset{x::s}{\rightarrow}$ is traversed when reading an input character x and it outputs the string s. (For further details, see Section 2.6.) For example, when applied to the string $s' = MM\$5\$5MM\$8$, the transducer outputs the string $s = MM55MM8$. Note that executing `escapeWS()` on s results in s', i.e.,

$$escapeWS^{-1}(MM\$5\$5MM\$8) = MM55MM8.$$

In fact, if we only consider strings comprised of characters coming from Γ then the transducer in Figure 2.1 can invert any string in the image of `escapeWS()`.

Static analysis for a restricted class of SMPs. One of the key technical contributions of this chapter is the design of a static analysis that can find a set of good characters for a given string manipulating procedure. Our analysis handles a restricted class of procedures. In this class, a procedure takes a string input and returns a string output. The procedure can read its input from left to right or from right to left, however each input character is read only once. The procedure is allowed to use variables that can hold character values and employ loops and conditions where a variable is compared with a constant character or another variable. While simple, we found that our restricted programming model is still expressive enough to handle a variety of procedures.

The analysis abstracts the execution trace of the procedure by maintaining an ordering over program variables according to the position of the input character that they got their value from. Essentially, whenever a variable containing an input character is written out of order, the analysis determines that the values of all the unwritten variables that may have been read before it are *not* good. Writing a constant character to the output also leads the analysis to decide that this character is not good either.

String inversion. Given the machine learning model approximating a pseudo inverse of $p()$ and an output string s', we first replace every good character in s' with the meta character. For example,

$$s' = Ab\$5\$5T@\$8 \xrightarrow{\text{translates to}} s'_0 = MM\$5\$5MM\$8\,.$$

We then execute the transducer using s'_0, which returns s_0. Recall that $escapeWS^{-1}(s'_0) = s_0$. Our main theorem (see Section 2.5) ensures that the static analysis indeed finds a set of good characters for the analyzed procedure. Thus, using Definition 2, we can get an input s which would lead to the desired output s' by replacing the meta character M back to the good character it came from in s'. For example,

$$s_0 = MM55MM8 \xrightarrow{\text{translates back to}} s = Ab55T@8.$$

Indeed, $escapeWS(Ab55T@8) = Ab\$5\$5T@\$8$.

Disclaimer. We note that our technique is not guaranteed to always find an input s which leads $p()$ to produce a particular output string s'. This can be because $p()$ is not surjective and $p^{-1}(s')$ is undefined, or because the translation model produced by the machine learning algorithm is not accurate enough. (To isolate any client application from this kind of inaccuracy, and as we have $p()$ at our disposal, we can execute $p()$ on

s and validate that indeed it returns s'). Furthermore, in case $p()$ is not injective, and there might be multiple inputs leading to a specific output, the model we learn may return an arbitrary string, which, untested by a forward execution, might not even be in the preimage of p. However, as our technique involves generating a random training set, re-executing the learning phase may create a different model which would possibly find a different input.

2.3 Programming Language

We formalize our results for a simple imperative programming language specialized for string processing: Every program receives a string as input and produces a string as output. The design of the language is inspired by real life examples of string manipulating procedures which often process their inputs character by character.[2]

Computation model. Roughly speaking, programs have at their disposal two *read heads* and two *write heads*. The input resides in the *read buffer*. The *first read head* is used to read the input from left to right, and the *last read head* allows to read the input from right to left. Once a read head inspects a character, it advances to the next position. A special built-in expression `done()` allows the programmer to determine whether all the input characters have been read. Trying to read a character after all the input has been read blocks the program.[3] The program produces its output using the write heads. The *first write head* writes characters to the program's *first write buffer*, and the *last write head* writes to the program's *last write buffer*. The first write buffer is written from left to tight and the last write buffer is written from right to left. When the program terminates, it returns a concatenation of the first and last write buffers. This model allows us to handle programs which process their input string in a character by character fashion and read every character at most once in a sequential manner going from the beginning of the string to it end, the other way around, or even alternating between the two directions.

[2]We remind the reader that our tool operates directly on LLVM bitcode.

[3]The choice to block the program was done in the sake of simplicity. Alternatively, we could have designed the language to signal an error.

$$\begin{array}{lcl}
\mathit{stmt} & ::= & \mathit{cmd} \;\mid\; \mathit{stmt}\ \mathit{stmt} \;\mid \\
 & \mid & \texttt{if}\ (\mathit{bexp})\ \{\ \mathit{stmt}\ \}\ \texttt{else}\ \{\ \mathit{stmt}\ \} \\
 & \mid & \texttt{while}\ (\mathit{bexp})\ \{\ \mathit{stmt}\ \} \\
\mathit{cmd} & ::= & x = \texttt{read-first}() \;\mid\; x = \texttt{read-last}() \\
 & \mid & \texttt{write-first}(x) \;\mid\; \texttt{write-last}(x) \\
 & \mid & x := \mathit{exp} \;\mid\; \texttt{return} \\
\mathit{exp} & ::= & \texttt{const} \mid y \\
\mathit{bexp} & ::= & x \bowtie \mathit{exp} \;\mid\; \texttt{done}() \;\mid\; \texttt{!done}()
\end{array}$$

```
d = $
while(!done()) {
    x = read-first()
    if (x = 5) {
        write-first(d) }
    if (x = 8) {
        write-first(d) }
    write-first(x) }
return
```

Figure 2.2: Syntax of the programming language and a version of procedure `escapeWS()` written in our programming language. $\bowtie \in \{=, \neq\}$

2.3.1 Syntax

Figure 2.2 defines the syntax of our programming language and, as an example, shows a possible encoding of procedure `escapeWS()` in our language.

A program is a statement *stmt*, which can be either a primitive command *cmd*, a sequential composition of statements, denoted by juxtaposition, a conditional statement, or a while loop.

Primitive commands allow to read input characters, write output characters, and assign the values of *expressions* to variables: The command $x =$ `read-first()` reads a character from the input using the first reading head, and assigns it to variable x. The command $x =$ `read-last`() does the same using the last read head. The commands `write-first(x)` and `write-last(x)` write the contents of x to the first and last output buffers, respectively. We allow for assignments of the form expression $x = \mathit{exp}$ where an expression exp is either a character variable or a constant character. The `return` command terminates the execution of the program and produces the output by concatenating the write buffers.

Conditional statements and while loops use boolean expressions *bexp* which allow to check whether the value of a given variable is equal to a given expression or not. Two additional boolean expressions are the special built in operators `done()` and `!done()`, which allow the program to determine whether all its input has, respectively, has not, been read.

$$
\begin{array}{lr}
(\sigma in, out_F, out_L, env) \xrightarrow{v=read-first()} (in, out_F, out_L, env[v \mapsto \sigma]) & \\
(in, out_F, out_L, env) \xrightarrow{write-first(v)} (in, out_F env(v), out_L, env) & \\
(in\sigma, out_F, out_L, env) \xrightarrow{v=read-last()} (in, out_F, out_L, env[v \mapsto \sigma]) & \\
(in, out_F, out_L, env) \xrightarrow{write-last(v)} (in, out_F, env(v) out_L, env) & \\
(in, out_F, out_L, env) \xrightarrow{v:=\texttt{const}} (in, out_F, out_L, env[v \mapsto \texttt{const}]) & \\
(in, out_F, out_L, env) \xrightarrow{v:=x} (in, out_F, out_L, env[v \mapsto env(x)]) & \\
(in, out_F, out_L, env) \xrightarrow{assume(x \bowtie \texttt{const})} (in, out_F, out_L, env) & env(x) \bowtie \texttt{const} \\
(in, out_F, out_L, env) \xrightarrow{assume(x \bowtie y)} (in, out_F, out_L, env) & env(x) \bowtie env(y) \\
(in, out_F, out_L, env) \xrightarrow{assume(done())} (in, out_F, out_L, env) & in = \epsilon \\
(in, out_F, out_L, env) \xrightarrow{assume(!done())} (in, out_F, out_L, env) & in \neq \epsilon \\
(in, out_F, out_L, env) \xrightarrow{return} out_F out_L &
\end{array}
$$

Figure 2.3: Concrete meaning of commands. $\bowtie \in \{=, \neq\}$

2.3.2 Concrete Semantics

Before defining the meaning of programs in our language, we introduce some notation.

Notation. We assume a (possibly infinite) domain (alphabet) of characters Σ ranged over by the meta variable σ, and a syntactic domain VAR of variable names, ranged over by $v, x, \ldots$, which we also use as a semantic domain. A *string* $s \in \overline{\Sigma}$ is a sequence of characters, i.e., a function from $1..n$, for some $n \in \mathbb{N}^0$, to Σ. In the following, we denote string concatenation by juxtaposition and the empty string by ϵ. Given a function env, we write $env[x \mapsto y]$ to denote a function which acts like env anywhere except at x, where its value is y.

Concrete memory states. A *concrete memory state*

$$m = (in, out_F, out_L, env) \in \mathcal{M} = \overline{\Sigma} \times \overline{\Sigma} \times \overline{\Sigma} \times E$$

is a quadruple. The first three components, namely in, out_F, and out_L, are strings which store the contents of the program's read buffer, first write buffer, and last write buffer, respectively. $env \in E = VAR \to \Sigma$ is an *environment* which records the values of variables. We assume that variables are initialized to some fixed *zero* character. By abuse of notation, we let $env(\texttt{const}) = \texttt{const}$ for any constant character $\texttt{const}$.

Operational semantics. Figure 2.3 defines the meaning of programs using a small step operational semantics. The latter is defined by translating the program into a control-flow graph form, and encoding conditional using *`assume`* commands in the standard way: executing an `assume(bexp)` command blocks the execution on state where `bext` does not hold, and does not change the state otherwise. The meaning of commands is rather self explanatory. We only direct the reader's attention to the fact that a `write-first()` command adds a character at the *end* of the first write buffer whereas the `write-last()` command adds a character at the *beginning* of the last write buffer and that the program gets stuck if it tries to read an input character when the read buffer is empty.

2.4 Instrumented Semantics

The purpose of our static analyzer is to help reduce the size of the input alphabet used by the machine learning algorithm when computing the pseudo-inverse of the analyzed SMP. To do so, it detects characters on which the SMP act as the identity function: It turns out that rather often a string-manipulating procedure treats many characters in a particularly uniform way; it only copies them once from the input to the output in an order-preserving fashion. The static analyzer conservatively finds these *good* characters, and enables the use of a single good representative character in the alphabet during learning. This reduction in the size of the alphabet translates to a huge benefit for the learning algorithms, as we discovered in our experiments.

The instrumented semantics extends the concrete one with properties which are of matter to the analysis. The main tracked property is the set $BAD \subseteq \Sigma$ of *bad* characters for the execution. We explain the role of BAD by describing its complement $GOOD = \Sigma - BAD$. A set of characters $G \subseteq \Sigma$ is *good* if every time a character $\in G$ is read off the input, it is copied as-is to the output. In particular, the subsequences of the input and output strings comprised of the good characters are identical. (See Definition 2.)

The goal of our static analysis is to determine a set of good characters for an SMP. The role of the instrumented semantics is to explicate which properties of the execution are tracked to facilitated this task. Thus, when the instrumented semantics terminates, it returns, in addition to the output string, the set of bad characters it computed.

```
    x = read-first()
    y = read-first()
3:  z = y
    if (y = $) { write-first(y); write-first(x) }
    else { write-first(x);  write-first(z) }
    while (!done()) { x = read-first();  write-first(x) }
```

Figure 2.4: SMP `showDough()`

Let $\hat{p}(s) = (s', BAD)$ denote the output $p()$ produces if it executes according to the instrumented semantics on input string s. The usefulness of the instrumented semantics as a basis for analysis stems from the following lemma.

Lemma 2. *Let $p()$ be an SMP and s, s' strings. If $\hat{p}(s) = (s', BAD)$ then $\Sigma \setminus BAD$ is a* good *set of characters for $p()$ and s.*

2.4.1 Instrumented States

Instrumented states record properties pertaining to the flow of information from the input string through variables to the output string. Specifically, every instrumented state augments a concrete state with four binary relations $E_F, E_L, R_F, R_L \subseteq \Sigma \times \Sigma$ and the set of possibly-not-good characters $BAD \subseteq \Sigma$. We refer to the quintuple $\iota = (E_F, E_L, R_F, R_L, BAD)$ as an *instrumentation*. We assume the components of the instrumentation are initialized to $\emptyset$.

$$\hat{m} = (m, \iota) \in \hat{\mathcal{M}} = \mathcal{M} \times \hat{\mathcal{I}}$$

$$\text{where} \quad \iota = (R_F, R_L, E_F, E_L, BAD) \in \hat{\mathcal{I}} = (2^{VAR \times VAR})^4 \times 2^{\Sigma} \, .$$

The m component of an instrumented state is the concrete state it augments.

E_F and E_L are equivalence relations over variable names. Recall that in our language, a variable can be assigned a value either by reading into it a character from the input, assigning into it a constant value, or assigning into it the value of another variable. E_F equates variables whose values originated from the same `read-first()` operation, either directly, or through a sequence of copy assignments. For example, in the instrumented state which arises at program point 3 in Figure 2.4, $E_F = \{(x, x), (y, y), (y, z), (z, y), \ (z, z)\}$. E_L does the same for variables whose value was originated from a `read-last()` command. In Figure 2.4, there are no `read-last()` commands. Thus, $E_L = \emptyset$ at any state which arises during the execution.

R_F and R_L are preorders over variable names. R_F represents the order in which variables were read using the first reading head, and R_F represents the order in which variables were read using the last reading head. Both orders take variable copy-assignments into account. For instance, at the instrumented state in program point 3, $R_F = \{(x,x),(y,y),\ (z,z),(y,z),(z,y),(x,y),(x,z)\}$. $R_L = \emptyset$ because there are no `read-last()` commands.

BAD over approximates the set of bad characters for the input string on which the SMP executes to produce the state. The over approximation is based on the flow of characters from the input string to output string, as we discuss in Section 2.4.2.

Healthiness conditions. The instrumentation in instrumented states respects certain natural *healthiness* conditions: A variable may appear in R_F only if it appears in E_F, as in a concrete execution the order in which input characters is read is total and every input character may be read at most once. In fact, R_F can be seen as a total order over the equivalence classes of E_F. A similar relation exists between R_L and E_L. Finally, a variable cannot appear in both E_F and E_L as an input character may be read either by the first read head or by the last read head.

2.4.2 Instrumented Small-Step Operational Semantics

The instrumentation is manipulated by the instrumented transformers presented in Figure 2.5 which defines a deterministic transition relation over $\hat{\mathcal{I}} \times \hat{\mathcal{I}}$.[4] The transition rules of the instrumented semantics extends the ones of the concrete semantics to track *must* value-flow information:

$$\frac{m \xrightarrow{cmd} m' \quad \iota \xrightarrow{cmd}_m \iota'}{(m,\iota) \xrightarrow{cmd} (m',\iota')}\ cmd \neq \texttt{return} \qquad \frac{m \xrightarrow{\texttt{return}} s \quad \iota \xrightarrow{\texttt{return}}_m BAD}{(m,\iota) \xrightarrow{\texttt{return}} (s, BAD)}$$

The transition relation of the instrumented part of the state is parameterized with the source concrete state of the transition because it requires access to the environment.

We define the rules in Figure 2.5, which we explain next, using the following shorthand: Let R resp. E be a binary resp. equivalence relation over vari-

[4]The transformers pertaining to `read-last()` and `write-last()` operations are similar to those of `read-first()` and `write-first()`, respectively, and are thus omitted.

$$(R_F, R_L, E_F, E_L, BAD) \xrightarrow{v=read-first()}_m (R^r_F, R_L|_{\neg v}, E_F|_{\neg v} \cup \{(v,v)\}, E_L|_{\neg v}, BAD \cup BAD^r)$$
$$R^r_F = R_F|_{\neg v} \cup \{(x,v) \mid (x,x) \in E_F\} \cup \{(v,v)\}$$
$$BAD^r = \begin{cases} \{env(v)\} & (v)_{E_F} = \{v\} \vee (v)_{E_L} = \{v\} \\ \emptyset & otherwise \end{cases}$$

$$(R_F, R_L, E_F, E_L, BAD) \xrightarrow{write-first(v)}_m \left(R^w_F, R_L|_{\neg v}, E_F|_{\neg((v)_{E_F})}, E_L|_{\neg((v)_{E_L})}, BAD \cup BAD^w\right)$$
$$R^w_F = \begin{cases} \{(z,y) \in R_F \mid z \notin (v)_{E_F}\} & (v,v) \in E_F \\ R_F & otherwise \end{cases}$$
$$BAD^w = \begin{cases} \{env(v)\} & (v,v) \notin E_F \vee (v,v) \in E_L \vee exp = \texttt{const} \\ \{\sigma \in \{env(y)\} \mid (y,y) \in E_F \wedge y \neq v \wedge (v,y) \notin R_F\} & (v,v) \in E_F \wedge exp = v \end{cases}$$

$$(R_F, R_L, E_F, E_L, BAD) \xrightarrow{v:=\texttt{const}}_m (R_F|_{\neg v}, R_L|_{\neg v}, E_F|_{\neg v}, E_L|_{\neg v}, BAD \cup BAD_v)$$
$$(R_F, R_L, E_F, E_L, BAD) \xrightarrow{v:=x}_m (R_F|_{\neg v}, R_L|_{\neg v}, E_F|_{\neg v}, E_L|_{\neg v}, BAD \cup BAD_v) \qquad (x,x) \notin E_F \cup E_L$$
$$(R_F, R_L, E_F, E_L, BAD) \xrightarrow{v:=x}_m (\mathrm{Add}(R_F|_{\neg v}, v, x), R_L, \mathrm{Add}(E_F|_{\neg v}, v, x), E_L, BAD \cup BAD_v) \qquad (x,x) \in E_F$$
$$(R_F, R_L, E_F, E_L, BAD) \xrightarrow{v:=x}_m (R_F, \mathrm{Add}(R_L|_{\neg v}, v, x), E_F, \mathrm{Add}(E_L|_{\neg v}, v, x), BAD \cup BAD_v) \qquad (x,x) \in E_L$$
$$\text{where } BAD_v = \begin{cases} \{env(v)\} & (v)_{E_F} = \{v\} \vee (v)_{E_L} = \{v\} \\ \emptyset & otherwise \end{cases}$$

$$(R_F, R_L, E_F, E_L, BAD) \xrightarrow{return}_m BAD$$

Figure 2.5: Instrumented semantics. The transformers pertaining to `assume` commands act like the identity function. $m = (\_, \_, \_, env)$. We assume $v \neq x$

able names and X a set of variable names. We use the following as shorthand $R|_{\neg X} \equiv \{(a,b) \in R \mid (a \notin X) \wedge (b \notin X)\}$ removes from R any pair coming from $X \times X$, $(v)_E \equiv \{x \mid (x,v) \in E\}$ denotes the equivalence class of v in E, and $\mathrm{Add}(R,v,x) \equiv R \cup \{(a,v) \mid (a,x) \in R\} \cup \{(v,a) \mid (x,a) \in R\}$ adds v to R in the same positions as x.

The instrumented semantics of a $v =$ `read-first`() command removes any mention of v from all the relations in the instrumentation—it might be there because its value could have come from a previous read command. It then places it in its own equivalence class in E_F and as the the minimal element in R_F: v is the only variable that got its value from that `read-first`() operation, which is the last command executed so far. If before the assignment v relates only to itself in either E_F or E_L then its value is about to be overridden and lost before having a chance to get written to the output. Hence, in this case the value of $env(v)$ is considered a bad character.

The instrumented meaning of `write-first`(v) removes any mention of v or any of the variables in its equivalence class according to E_F from any relation it belongs to. This is because a *read* good character should not be written more than one time to the output. If v got its value from a constant assignment or from the opposite read head, or if its value has already been written then $env(v)$ becomes a bad character. If v did get its value from the *first-head* but it is not written in the right order, i.e., it is not a maximal element in R_F then the contents of all the larger variables in R_F becomes bad.[5]

The instrumented meaning of a `v` $:=$ *exp* removes v from its current place in the instrumentation and, if *exp* is a variable, places v in the same relations and at the same positions as *exp*. If v was the only variable to contain a value coming from a read command then $env(v)$ becomes a bad character.

The instrumented meaning of a `return` command ends the execution with the accumulated BAD set.

The rules in Figure 2.5 never interfere with neither the values nor the control of the underlying concrete semantics. They also preserve healthiness.

Lemma 3. *Let (m, ι) and (m', ι') be instrumented states and cmd a command such that $(m, \iota) \xrightarrow{cmd} (m', \iota')$. If ι is a healthy instrumentation then ι' is healthy too.*

2.5 Static Analysis

Our abstract interpretation algorithm over-approximates the instrumented semantics described in Section 2.4 by replacing the concrete memory state component of instrumented states with an abstract one.

Abstract States Our static analysis algorithm computes an *abstract instrumented state*

$$A = (m^\sharp, \iota) \in \mathcal{A} = \mathcal{M}^\sharp \times \hat{\mathcal{I}} \qquad \text{where } \; m^\sharp = (Done, env^\sharp) \in \mathcal{M}^\sharp\,.$$

at every program point. An abstract instrumented state is comprised of an *abstract state* $m^\sharp = (Done, env^\sharp)$ and an instrumentation $\iota \in \hat{\mathcal{I}}$ (see Section 2.4.1). The *Done* component of the abstract state abstracts the number of unread characters, i.e., whether the two read-heads passed each other or not: $\{T\}$ means that all the input characters have been read, $\{F\}$ means the opposite, and $\{T, F\}$ means that the situation is unknown. $env^\sharp : VAR \to 2^\Sigma$ is an abstract environment mapping variable names to the sets of their possible values. The instrumentation component ι is utilized for the same purpose and in the same way as in the instrumented semantics.

[5] An equally plausible alternative would be to make $env(v)$ bad.

Notice that while $env^\sharp(x) \in 2^\Sigma$ may be infinite, the only changes to it are additions and removals of values that occur as literal constants in the program. Therefore the number of such distinct values is at most 2^k, where k is the number of such constants. This provides a termination guarantee of the analysis even with an infinite alphabet.

Join. The least upper bound (join) operator is defined as follows:

$$(m_1^\sharp, \iota_1) \sqcup (m_2^\sharp, \iota_2) = (m_1^\sharp \sqcup m_2^\sharp, \iota_1 \sqcup \iota_2)\,, \qquad \text{where}$$

$$(Done_1, env_1^\sharp) \sqcup (Done_1, env_1^\sharp) = (Done_1 \cup Done_1, \lambda x.\ env_1^\sharp(x) \cup env_2^\sharp(x))$$

$$\left(R_F^1, R_L^1, E_F^1, E_L^1, C^1, BAD^1\right) \sqcup \left(R_F^2, R_L^2, E_F^2, E_L^2, C^2, BAD^2\right) =$$

$$(R_F^1 \cap R_F^2, R_L^1 \cap R_L^2, E_F^1 \cap E_F^2, E_L^1 \cap E_L^2, BAD^3)$$

$$\text{where}\ \ BAD^3 = BAD^1 \cup \{\sigma \in \rho_1(x) \mid x \in E_F^1 - E_F^2 \cup E_L^1 - E_L^2\} \cup$$

$$BAD^2 \cup \{\sigma \in \rho_2(x) \mid x \in E_F^2 - E_F^1 \cup E_L^2 - E_L^1\}$$

With the exception of the BAD component, it is easy to see that the the resulting state is indeed the least upper bound of the two abstract instrumented states. The reason we chose in to intersect most of the component of joined instrumentations is rather clear—we track must information. To understand the reason why defining $BAD^3 = BAD^1 \cup BAD^2$ would not suffice to ensure a sound analysis consider a scenario when $(x, x) \in E_F^1 - E_F^2$. Had we kept $(x, x) \in E_F^1 \cap E_F^2$ $(:= E_F^3)$, then a future `write-first`(x) possibly violates the goodness of the character set $\rho_1(x) \cap \rho_2(x)$ $(:= \rho_3(x))$ as it may be written without ever being read. On the other hand, as we discarded (x, x) from E_F^3, we opened the door for a future `x` $=$ `read-first`$()$ to possibly violate the goodness of the character set $\rho_3(x)$, as some characters may have been read without ever being written. So whenever $(x, x) \in E_F^i - E_F^j$ $(\{i, j\} = \{1, 2\})$ we include $\rho_i(x)$ in BAD^3. The same line of reasoning applies to E_L too. Thus, we add to BAD^3 the characters associated with the variables found in the symmetrical difference of the relevant equivalence realtions.

2.5.1 Concretization

The concrete domain which we use to justify the soundness of our analysis is the powerset of instrumented states. The concretization function γ maps an abstract state

$$(env^\sharp, Done) \xrightarrow{v=read-first()\ /\ v=read-last()} (env^\sharp[x \mapsto \Sigma], Done \cup \{T\}) \qquad Done \neq \{T\}$$

$$(env^\sharp, Done) \xrightarrow{write-first(v)\ /\ write-first(v)} (env^\sharp, Done)$$

$$(env^\sharp, Done) \xrightarrow{v:=\texttt{const}} (env^\sharp[v \mapsto \{\texttt{const}\}], Done)$$

$$(env^\sharp, Done) \xrightarrow{v:=x} (env^\sharp[v \mapsto env^\sharp(x)], Done)$$

$$(env^\sharp, Done) \xrightarrow{assume(v=\texttt{const})} (env^\sharp[v' \mapsto \{\texttt{const}\} \mid \varphi(v)], Done) \qquad \texttt{const} \in env^\sharp(x)$$

$$(env^\sharp, Done) \xrightarrow{assume(v!=\texttt{const})} (env^\sharp[v' \mapsto env^\sharp(x) \setminus \{\texttt{const}\} \mid \varphi(v)], Done) \qquad \{\texttt{const}\} \neq env^\sharp(x)$$

$$(env^\sharp, Done) \xrightarrow{assume(v=x)} (env^\sharp[v', x' \mapsto env^\sharp(v) \cap env^\sharp(x) \mid \varphi(v) \wedge \varphi(x)], Done) \qquad env^\sharp(v) \cap env^\sharp(x) \neq \emptyset$$

$$\begin{array}{ll}(env^\sharp, Done) \xrightarrow{assume(v!=x)} (env^\sharp[v \mapsto V, x \mapsto X \mid \varphi(v) \wedge \varphi(x)], Done) & env^\sharp(v) = env^\sharp(x) \\ \quad V = \text{if } (|env^\sharp(x)| = 1) \text{ then } env^\sharp(v) \setminus env^\sharp(x) \text{ else } env^\sharp(v) & \implies |env^\sharp(v)| > 1 \\ \quad X = \text{if } (|env^\sharp(v)| = 1) \text{ then } env^\sharp(x) \setminus env^\sharp(v) \text{ else } env^\sharp(x) & \end{array}$$

$$(env^\sharp, Done) \xrightarrow{assume(done())} (env^\sharp, \{T\}) \qquad \{F\} \neq Done$$

$$(env^\sharp, Done) \xrightarrow{assume(!done())} (env^\sharp, \{F\}) \qquad \{T\} \neq Done$$

$$(env^\sharp, Done) \xrightarrow{return} (env^\sharp, Done))$$

Figure 2.6: Abstract semantics. $\varphi(z) = z' = z \vee z' \in (z)_{E_F} \vee (z)_{E_L}$

to a set of instrumented ones. Let $\iota = (R_F, R_L, E_F, E_L, BAD)$, then

$$\begin{aligned}\gamma(((Done, env^\sharp), \iota)) = \{((in, out_F, out_L, env), (R^c_F, R^c_L, E^c_F, E^c_L, BAD^c)) \mid \\ in = \epsilon \to T \in Done \wedge in \neq \epsilon \to F \in Done \wedge \\ \forall x.\ env(x) \in env^\sharp(x) \wedge \\ R^c_F \supseteq R_F \wedge R^c_L \supseteq R_L \wedge E^c_F \supseteq E_F \wedge E^c_L \supseteq E_L \wedge BAD^c \subseteq BAD\end{aligned}$$

When an abstract instrumented state $\iota^\sharp = (A, \iota)$ represents an instrumented state $((in, out_F, out_L, env), \iota^c)$, A's *Done* component conservatively tracks whether all the input characters in *in* have been read and that the values *env* gives to variables agree with the ones provided by the abstract environment. The instrumentation ι^c of the concrete state should track no less information regarding the information flow of characters from the input string to the output string as does the instrumentation ι. The latter should also consider as bad any bad character in ι^c.

2.5.2 Abstract Transformers

The abstract transformers are defined by replacing the concrete component in the transition rules of the instrumented semantics with the rules pertaining to abstract states defined in Figure 2.6 and adapting the rules in Figure 2.5 to utilize an abstract environment instead of a concrete one as explained below

$$\frac{m^\sharp \xrightarrow{cmd} m^{\sharp\prime} \qquad \iota \xrightarrow{cmd}_{m^\sharp} \iota'}{(m^\sharp, \iota) \xrightarrow{cmd} (m^{\sharp\prime}, \iota')}$$

Again, the transition relation of the instrumented part of the state is parameterized with the source abstract state of the transition because it requires access to the abstract environment. The required adaptation of Figure 2.5 is rather direct: where ever an expression of the form $\{env(x)\}$ appears in a rule, we replace it with $env(x)$.

The rules are quite simple; the tricky ones pertain to `assume` statements regarding inequalities, which we now explain.

The abstract transformer of command `assume`(v!=`const`) blocks the execution if the only possible value of v is `const`. Otherwise, it merely records that `const` is not in fact a possible value of v. Note that not only $env^\sharp(v)$ may be adapted, but in case v got its value from the input string, any variable who got its value from the same read operation, i.e., in the same equivalence class as v in either E_F or E_L, may have the set of its possible values refined.

The abstract transformer of command `assume`(v!=`const`) blocks the execution if the only possible value of v is `const`. Otherwise, it merely records that `const` is not in fact a possible value of v. The abstract transformer of command `assume`(v!=y) blocks the execution if the abstract environment associates both variables with the same single character. Otherwise it attempts to refine the set of possible values of one variable if the other one is associated with a singleton set.

Main Theorem The static analysis algorithm computes at every program point an abstract state which over-approximates any instrumented state which can arise at this point for some input string. We denote by $BAD(p)$ the union of the BAD sets at $p()$'s exit points, i.e., right after $p()$ executes a return command. Our main theorem, whose proof follows directly from Lemma 2 and the soundness of the analysis, states that the analyzer computes a set of good characters $p()$.

Theorem 1. *Let $p()$ be an SMP. $\Sigma \setminus BAD(p)$ is a* good *set of characters for $p()$.*

2.6 Learning Pseudo-Inverse Functions

Our overall goal is that given an SMP p and a desired output string s' to find a string s such that $p(s) = s'$. One natural candidate for s is s' itself. Thus, when trying to learn an inverse we look for an input $s \neq s'$ such that $p(s) = s'$ and hence when generating the training examples, we only use ones where the input is different from the output. Also, if $p()$ is not injective, then it may have many pseudo-inverses, and there is no a priori way to favor any of them. Thus, it suffices to learn an *arbitrary* pseudo-inverse of $p()$.

The learning algorithms chosen to be employed in this paper are the ones we thought handle best the SMPs we have examined. However, they can be easily interchanged with others—our approach, as we said before, is indepedent of the chosen learning algorithm.

2.6.1 Learning Transducers with OSTIA

Transducers are deterministic finite state machines that are used to translate strings. We explain them using the example transducer depicted in Figure 2.7. Just like DFAs, transducer read their input strings from the left to the right, character by character, and traverse edges according to a transition function. In addition, as edges are traversed, the transducer prints characters to the output. If the input string $MMMM\#\#$ is fed to the transducer, it will go through states $0, 2, 5, 0, 2, 6, 0$, and print the output string $MMMM\#\$\#$. Any states of the transducer can hold inner strings. If some state q is a final state for the transduction, and it holds an inner string *sigma*, then it is appended at the end of the output. For example, the transduction of $MMMM\#$ equals $MMMM\$\#$.

OSTIA [82] is a supervised learning algorithm that is capable of learning transducers. Assuming the training set is without noise, like in our case, OSTIA is guaranteed to converge to the real transducer as the size of the training set increases. The SMP from Figure 2.2 and its inverse are both transducers, and we depict them in Figures 2.1 and 2.8, respectively. Every state of the transducer is depicted as a square containing a unique identifier, with state 0 being the initial state, and the string *sigma* which is appended to the output if the transduction end at that state. Transitions between states are depicted as edges annotated with $\sigma :: s$ denoting the character σ which triggers the

```
char* ReplaceSpaces(char *in) {
  char *out = malloc(MAXLEN);
  char *s = out;
  char c = 0;
  char skip_next_lf = 0;
  while (*in != 0) {
    c = *in; in++;
    if (skip_next_lf) {
      skip_next_lf = 0;
      if (c == '#') {
        c = *in; in++;
        if (!c) break; }}
    if (*c == '$') {
      skip_next_lf = 1;
      c = '#'; }
    *s = c; s++;}}
  *s = *in;
  return out;}
```

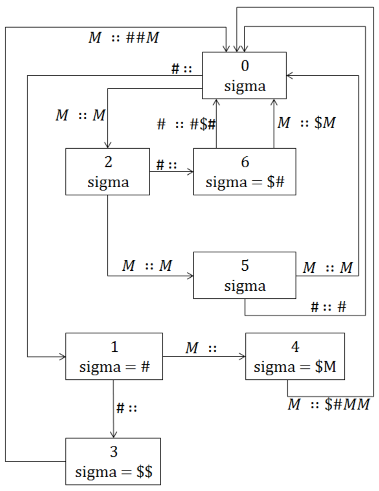


Figure 2.7: An SMP written in C and its pseudo-inverse as learned by OSTIA.

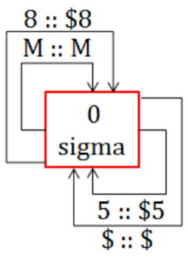


Figure 2.8: A transducer implementing the SMP in Figure 2.2

transition and the string s appended to the output due to taking the transition.

OSTIA succeeds in learning the exact inverse at the righthand side of Figure 2.7. In its essence, OSTIA is an iterative state merging algorithm. At each step the algorithm considers pairs of states as candidates for a possible merge, and if the resulting merged transducer is consistent with the training set, it accepts the merge and proceeds to the next iteration. The transducer in Figure 2.7 is the pseudo inverse OSTIA learns for the SMP shown in the same figure. The character # in an output string could have originated from either #, \$ or \$# in the input string. While randomizing our input for the training set, all three possibilities introduce themselves. This is evident in the transducer, as the edges $(5, 0), (2, 6), (6, 0)$ choose a different source for the # character each. Thus neither # nor \$ can be good characters.

2.6.2 Needleman Wunsch Alignment Algorithm

To show the versitility of our approach, we also used the alignment algorithm of Needleman-Wunsch [106] to learn procedure inverses. The algorithm is designed to align input and output strings, where the latter comes from the former by performing a sequence of steps. In each step a character is either deleted, inserted or replaced by another character. Naturally, as the number of steps is smaller, and the input and output are close in terms of edit distance, the results of the alignment are better. Our application uses a random set of inputs $\{s_i\}_{i=1}^{n}$ just as before, and apply p on each element of the set to end up with a training set $\{(s_i, p(s_i))\}_{i=1}^{n}$. Note, the order of the training set has changed, as we now want to learn the effect of the *original* SMP p. Each (input,output) pair is then aligned, and three probability tables are accumulated for the original SMP p: (1) A two dimensional table $T_r(p)$ for character replacements, in which $T_r(p)[\$][\#] = 0.45$ means that there is an estimated probability that a $ in the input string will be replaced by a #. (2) A one dimensional table $T_d(p)$, in which $T_d(p)[*] = 0.95$ means there is a probability of 0.95 that $*$ will be deleted from the input string. (3) A one dimensional table $T_i(p)$, in which $T_i(p)[@] = 0.55$,means there is a 0.55 probability of inserting @ *somewhere* in the output. Once these tables have been learned for the original SMP p, they can be used to deduce pseudo inverses p^{-1}: If $T_r(p)[\$][\#] = 0.45$ then clearly $T_r(p^{-1})[\#][\$] = 0.45$. Deducing $T_i(p^{-1})[*]$ based on $T_d(p)[*]$ is a little more subtle, and should also take into account the prevalence of the character $*$ in the input strings of the training set. Note that for more accurate results, $T_i(p^{-1})[*]$ depends on the length of the string y it wishes to invert. Finally, computing $T_d(p^{-1})[@]$ from $T_i(p)[@]$ depends on the prevalence of the character @ in the output strings of training set, the prevalence of @ in y, and the length of y too. It is important to stress out that the resulting pseudo inverse p^{-1} is *not* deterministic, and could return different outputs when invoked multiple times. This can be seen as an advantage, because of $p^{-1}(y)$ failed to find a relevant x, we do not have to perform the learning process again, but simply call $p^{-1}(y)$ again.

2.7 Implementation and Experimental Evaluation

We have implemented our ideas in a tool called StrInver. The tool gets as input an SMP $p()$ written in LLVM [96] intermediate representation language, and a concrete query string y. (In our experiments, we used procedures written in C, compiled using Visual C 2010.) The tools checks whether the procedure falls within the class of restricted SMPs we handle (see Section 2.3) by expecting it to follow certain syntactic conventions, and if so it looks for a string s such that $s \neq s'$ and $p(s) = s'$. If the learning algorithm failed to find a model that returns a non-identity inverse for the given string s', it is retried with a new randomized training set. The algorithms were trained using a training set comprised of 64-100 examples, with a bias towards choosing shorter strings.

Table 2.1 summarizes our experimental results. We considered four string manipulating procedures coming from real-life software. `DPSTrim()` removes prefixes and suffixes comprised of character #. It is taken from *DataparkSearch* [1] open source search engine and is used to help parse configuration files. `escapeWS()` is our running example shown in Figure 2.1 and `ReplaceSpaces()` is shown in Figure 2.7. Both come from GCC. `DosNames()` is a python library function which replaces all the dots in a file name with underscores, except for the last one.

In our experiments, we randomly chose output strings using uniform distribution and with average length of 32 characters. We applied our technique to invert 100 strings for and each procedure. Table 2.1 shows the reduced alphabet our analysis discovered and the machine learning algorithm which we used. We ran our experiments in a laptop equipped with an $i5$ $2.3Ghz$ CPU with 6GB memory running Windows 7. In all our experiments, it took our analysis to invert each string less than 10 seconds, whereas KLEE [27], a state of the art symbolic executor, failed to invert any string after running for one hour. (KLEE was able to invert short strings containing around 5 characters in a few seconds.) Similarly, a machine learning algorithm trained with randomly selected strings chosen using the full alphabet failed to invert the given output strings. It might be the case that using a larger training set would make the naive machine learning more successful, however, this process might lead to expensive analyses as the space of possible strings grows exponentially with the length of the string.

2.8 Related Work, Conclusion and Future Work

Automatic inversion of programs was first studied by Dijkstra who manually inverted simple array-manipulating programs [41]. Follow up works looked at inverting (i) simple

Procedure	Reduced alphabet	ML. Alg.
DPSTrim()	$\{M, \#\}$	Needleman Wunsch
escapeWS()	$\{\$, 5, 8, M\}$	OSTIA
ReplaceSpaces()	$\{\$, \#, M\}$	OSTIA
DosNames()	$\{., \_, M\}$	OSTIA

Table 2.1: Experimental evaluation of selected SMPs. The table shows the size of the reduced alphabet and the machine learning algorithm used to model the pseudo inverse.

programs whose semantics is given as logic programs [120], (ii) tree-traversal programs using relational calculus and deductive methods [29, 128], (iii) array transformers using techniques based on LR-parsing [55, 87] or testing [84], and (iv) bijective string-manipulating procedures [75, 101]. To the best of our knowledge, we are the first to apply machine learning tools to invert programs. We also note that the programs we invert are not necessarily injective.

Recent advances in machine learning lead researchers to explore its capabilities in helping challenging program analysis tasks, e.g., specification inference [118, 124], speed up abstraction refinement [60], invariant generation [52, 108, 123], setting up parameters for parametrized static analyses [110], and infer clustering of variables in partially relational static analyses [70]. In our work, we address a dual question–how can machine learning technique help program analyses. To the best of our knowledge the question has not been widely addressed, with the notable exception of [107] which also argues that a combination of machine learning and program analysis can be a win-win situation.

Another active research area is the use of input/output examples to learn computer programs. Often, this is done in the context of synthesis, where examples guide a search-based synthesis process [63]. For example, in [62], a learning procedure is used to synthesize string manipulating procedures which appears in the context of spreadsheets based on syntactic manipulation. Another attack on this problem was taken in [131], where the procedures were synthesized using database-like lookup operations. In these works, the focus is on designing a language in which programs can be synthesized and an efficient search heuristics. In this work we too focus on string manipulating procedures (SMPs), which are abundant in almost all software packages. However, instead of asking the user for input/output examples, we analyze the code of one procedure and its behavior, as expressed by input-output pairs, to synthesize another procedure.

In [152], the authors suggest to learn the behavior of a procedure by inspecting its code and input-output examples. Their technique applies to a class of procedures which accepts their input character by character, e.g., multi-digit addition. They use recurrent neural network models with long short-term memory to accurately learn a model of the procedure behavior as a sequence-to-sequence transformer [141]. It can be interesting to see if a preliminary phase of program analysis, as we do in this work, can help improve the accuracy of their technique.

String solvers, e.g., [16, 39, 88], can reason about constraints involving operations on strings. For example, HAMPI [88], can reason about constraints expressing membership in regular languages and fixed-size context-free languages. In contrast, we provide a technique based on a combination of machine learning and static analysis that can help invert string manipulating procedures written in a restricted programming language.

Conclusion and Future Work We present a machine learning-based approach for inverting string manipulating procedures (SMPs). To the best of our knowledge, the use of machine learning for program inversion is novel. We make the approach feasible by developing a static analysis which reduces the size of the alphabet of the examples used during training. While the idea of reducing the input domain size is a known idea, we believe that we are the first to design a static analysis specific for enabling such a reduction. We evaluated our technique using a small selection of procedures taken from real-life software. Our approach does not require that the inverted SMP be bijective. However, our analysis is beneficial when the SMP acts as the identity on a large part of its alphabet, which we refer to as the “good” characters.

Chapter 3

String Length Reduction for Verifying Function Equivalence

This chapter is based on the results published in [85].

Strings are perhaps the most widely-used datatype, at the core of the interaction between humans and computers, via command-line utilities, text editors, web forms, and many more. Therefore, most programming languages have strings as a primitive datatype, and many program analysis techniques need to be able to reason about strings to be effective.

Recently, the rise of string solvers [16, 18, 39, 88, 143, 154] has enabled more effective program analysis techniques for string-intensive code, *e.g.* Kuduzu [125] for JavaScript or Haderach [129] and JST [54] for Java. However, these techniques operate on domains where strings are well-defined objects (*i.e.* the *String* class in Java). Unlike Java or similar languages, strings in C are just arbitrary portions of memory terminated by a null character. That means interacting with strings does not have to go through a well-defined API as in other programming languages. While there are string functions in the C standard library (e.g. *strchr*, *strspn*, etc.defined in *string.h*), programmers can—and as we will show often do—write their own equivalent loops for the functionality provided by the standard library.

In this paper, we focus on a particular set of these loops, which we call *memoryless loops*. Essentially, these are loops that do not carry information from one iteration to another. To illustrate, consider the loop shown in Figure 3.1 (taken from *bash* v4.4). This loop only looks at the current pointer and skips the initial whitespace in the string

```
#define whitespace(c) (((c) == '␣') || ((c) == '\t'))
char* loopFunction(char* line)
{
    char *p;
    for (p = line; p && *p && whitespace (*p); p++);
    return p;
}
```

Figure 3.1: String loop from the *bash v4.4* codebase, extracted into a function.

line. The loop could have been replaced by a call to a C standard library function, by rewriting it into `line += strspn(line, "␣\t")`. [1]

Replacing loops such as those of Figure 3.1 with calls to standard string functions has several advantages. From a software development perspective, such code is often easier to understand, as the functionality of standard string functions is well-documented. Furthermore, such code is less error-prone, especially since loops involving pointer arithmetic are notoriously hard to get right. Our technique is thus useful for refactoring such loops into code that calls into the C standard library.

From a program analysis perspective, reasoning about calls that use standard library functions is easier because the semantics is well-understood; conversely, understanding custom loops involving pointers is difficult. Moreover, code that uses standard string functions can benefit from recent support for string constraint solving. Solvers such as HAMPI [88] and Z3str [18, 153, 154] can effectively solve constraints involving strings, but constraints have to be expressed in terms of a finite vocabulary that they support. Standard string functions can be easily mapped to this vocabulary, but custom loops cannot. In this paper, we show that program analysis via dynamic symbolic execution [26] can have significant scalability benefits by using a string solver, with a median speedup of 79x for the loops we considered.

Finally, translating custom loops to use string functions can also impact native execution, as such functions can be implemented more efficiently, *e.g.* to take advantage of architecture-specific hardware features.

The goal of our technique is to translate[2] loops such as the one in Figure 3.1 into calls to standard string functions.

[1]We remind the reader that `strspn(char *s, char *charset)` computes the length of the prefix of `s` consisting only of characters in `charset`.

[2]In this paper, we use the terms translate, summarise and synthesise interchangeably to refer to the process of translating the loop into an equivalent sequence of primitive and string operations.

3.1 Loops Targeted

Our approach targets relatively simple string loops which could be summarised by a sequence of standard string library calls (such as `strchr`) and primitive operations (such as incrementing a pointer). More specifically, our approach targets loops that take as input a pointer to a C string (so a `char *` pointer), return as output a pointer into that same string, and have no side effects (e.g., no modifications to the string contents are permitted). Such loops are frequently coded in real applications, and implement common tasks such as skipping a prefix or a suffix from a string or finding the occurrence of a character or a sequence of characters in a larger string. Such loops can be easily synthesised by a short sequence of calls to standard string functions and primitive operations. While our technique could be extended to other types of loops, we found our choice to provide a good performance/expressiveness trade-off.

More formally, we refer to the two types of loops that we can synthesise as *memoryless forward loops* and *memoryless backward loops*, as defined below.

Definition 1 (Memoryless Forward Loop)**.** *Given a string of length len, and a pointer p into this buffer, a loop is called a forward memoryless loop with cursor p iff:*

1. *The only string location being read inside the loop body is $p_0 + i$, where p_0 is the initial value of p and i is the iteration index, with the first iteration being* 0
2. *Values involved in comparisons can only be one of the following: 0, i, len for integer comparisons; p_0, $p_0{+}i$, $p_0{+}len$ for pointer comparisons; and $*p$ (i.e. $p_0[i]$) and constant characters for character comparisons;*
3. *The conceptual return value (i.e. what the loop computes) is $p_0 + c$, where c is the number of completed iterations.*[3]

Definition 2 (Memoryless Backward Loop)**.** *Such loops are defined similarly to forward memory loops, except that the only string location being read inside the loop body is $p_0 + (len - 1) - i$ and the return value is $p_0 + (len - 1) - c$.*

We collaborated with Kapus et al. which developed an enumerative synthesizer comprising of string functions composed together with string atoms like pointer increment, null checking etc. The synthesised programs were only symbolically tested to

[3]I.e., the number of times execution reached the end of the loop body and jumped back to the loop header [5].

have the same effect as the original loop *for all strings up to a given bound.* In this section, we show how we can lift these bounded checks into a proof of equivalence.

Intuitively, we show that the loops we target cannot distinguish between executing on a "long" string and executing on its suffix. In both executions, the value returned by the loop is uniquely determined by the number of iterations it completed, and every time the loop body operates on a character c, it follows the same path, except, possibly, when scanning the first or last character.

Technically, we define a syntactic class of *memoryless specification* (Definition 3), which scans the input string either forwards or backwards, and terminates when it reaches a character which belongs to a given set X. The choice of this class of specification is a pragmatic one: We observed that all the programs we synthesise can be specified in this manner.

We then prove that if an arbitrary loop respects a memoryless specification on all strings up to length 3 and the original loop also adheres to certain easy-to-verify mostly-syntactic restrictions, the loop respects the specification on arbitrary strings. Essentially, we prove a small-model theorem: We show that any divergence from the specification on a string of length $k+1$, for $3 \leq k$, can be obtained on a string of length k.

The proof goes in two stages. First, we prove a small-model theorem (Theorem 3) for a class of programs (memoryless loops) which is defined semantically. For example, the definition restricts the *values* the loop may use in comparisons. Second, §3.4 shows most of our programs respect certain easy-to-check properties. The combination of the aforementioned techniques allows us to provide a conservative technique for verifying that the synthesised program is equivalent to the original loop.

Notations. We denote the domain of characters by $\mathcal{C}$ and use $\mathcal{C}_0 = \mathcal{C} \cup \{null\}$ for the same domain extended with a special *null* character. We denote (possibly empty) sequences of non-*null* characters by $\mathcal{C}^*$. We refer to a *null*-terminated array of characters as a *string buffer* (*string* for short). We denote the set of strings by $s \in \mathcal{S}$. We write constant strings inside quotation marks. For example, the string $s =$ `"abc"` is an array containing *four* characters: 'a', 'b', 'c', and *null.* Note that if $\omega = abc$ then $s =$ `"`ω`"`.[4] We use `""` to denote an *empty string*, i.e., one which contains only the *null* character.

[4]ω is a mathematical sequence of characters, s is a string buffer.

The *length* of a string s, denoted by `strlen(`s`)`, is the number of characters it contains excluding the terminating *null* character. For example, `strlen("abc")` $= |abc| = 3$ and `strlen("")` $= 0$. We denote the i^{th} character of a string s by $s[i]$. Indexing into a string is zero-based: $s[0]$ is the first character of s and $s[$`strlen(`s`)`$]$ is its last. Note that the latter is always *null*. We write "$c\omega$" resp. "ωc" to denote a string whose first resp. penultimate character is c. We denote the complement of a set of characters X by $\overline{X} = \mathcal{C}_0 \setminus X$.

3.2 Memoryless Specifications

Definition 3 (Memoryless Specification)**.** *A* memoryless specification *of a string operation is a function whose definition can be instantiated from the following schema by specifying the missing code parts (start, end, R, and X):*

```
char* func(char *input) {
  int i, len = strlen(input);
  for (i = start to end)
    if (input[i] ∈ X)
      return input + i;
  return R;
}
```

The schema may be instantiated to a function that traverses the input buffer either forwards *or* backwards*: In a forward traversal, start = 0, end =* `len - 1`*, and R =* `input + len`*. In a* backward *traversal, start =* `len - 1`*, end = 0, and R =* `input`*. X is a set of characters.*

Example 1. *It is easy to see that many standard string operations respect a memoryless specification. For example, the loop inside* `strchr(p,c)` *resp.* `strrchr(p,c)` *can be specified using a forward resp. backward memoryless specification with X set to $\{c\}$. The loop inside* `strspn(p,s)` *has a memoryless forward specification in which X contains all the characters* except *the ones in* `s`*.*

In this section we focus exclusively on memoryless forward loops. The definitions and proofs for memory backward loops are analogous, and are thus omitted.

We use $[\![P]\!]$ for the semantic function of P, that is, $[\![P]\!](s)$ is the value returned by P when its string buffer is initialised to $s \in \mathcal{S}$. We refer to the character that a memoryless forward loop may observe in the i^{th} iteration as the *current character of iteration* i, and omit the *iteration index* i when clear from context.

Definition 4 (Iteration Counter). *Let P be a memoryless forward loop, as per Definition 1. We define $\Delta_P(s)$ as the number of iterations that P completes when its input is a string s. If P does not terminate on s, then $\Delta_P(s) = \infty$.*

Note that the semantic restrictions imposed on memoryless loops ensures that $[\![P]\!]$ and Δ_P have a one-to-one mapping. Thus, in the rest of the paper, we use these notations interchangeably.

3.3 Bounded Verification of Memoryless Equivalence

Theorem 1 (Memoryless Truncate). *Let P be a memoryless forward loop, and let $\omega, \omega' \in \mathcal{C}^*$.*

1. *If $\Delta_P(\texttt{"}\omega\omega'\texttt{"}) < |\omega|$, then $\Delta_P(\texttt{"}\omega\omega'\texttt{"}) = \Delta_P(\texttt{"}\omega\texttt{"})$.*
2. *If $\Delta_P(\texttt{"}\omega\omega'\texttt{"}) \geq |\omega|$, then $\Delta_P(\texttt{"}\omega\texttt{"}) \geq |\omega|$.*

Proof. 1. Since P performs fewer than $|\omega|$ complete iterations, Definition 1 ensures that P can only observe the prefix $0..(|\omega| - 1)$ of its input buffer $\texttt{"}\omega\omega'\texttt{"}$,[5] which are all characters of ω. Moreover, all comparisons between integers $0, i, len$ or pointers $p_0, p_0 + i, p_0 + len$ must return identical values whether $len = |\omega|$ or $len = |\omega\omega'|$, since $i < |\omega|$ (and thus $i < |\omega\omega'|$) in all of these iterations. Therefore P behaves the same when executing on ω and on $\omega\omega'$, and thus it must be that $[\![P]\!](\texttt{"}\omega\texttt{"}) = [\![P]\!](\texttt{"}\omega\omega'\texttt{"})$.

2. Similarly, the first $|\omega|$ iterations are identical between $[\![P]\!](\texttt{"}\omega\omega'\texttt{"})$ and $[\![P]\!](\texttt{"}\omega\texttt{"})$. Since the former carried these $|\omega|$ iterations to completion, so must the latter perform at least as many iterations. □

Note that if P is *safe*, that is, never reads past the end of the string buffer, then $\Delta_P(\texttt{"}\omega\texttt{"}) \geq |\omega|$ in fact implies $\Delta_P(\texttt{"}\omega\texttt{"}) = |\omega|$. At this point we make no such assumptions; later we will see that if P is tested against a specification which is itself safe, then indeed it is guaranteed that P is also safe.

[5]When P performs k complete iterations, then it can read at most $k + 1$ (rather than k) characters from the input string. The last one occurs in the (incomplete) iteration that exits the loop.

Let $Q_0(c)$ denote whether P completes (at least) the first iteration of the loop when it executes on some single-character string "c", i.e., $Q_0(c) \triangleq (\Delta_P(\texttt{"}c\texttt{"}) > 0)$.

If $Q_0 = \mathit{false}$, i.e., $\forall c \in \mathcal{C}. \neg Q_0(c)$, then obviously P never gets to the second iteration of the loop. Otherwise, let $a \in \mathcal{C}$ be a character such that $Q_0(a)$ is true. We can continue to define Q_1 as:

$$Q_1(c) \triangleq (\Delta_P(\texttt{"}ac\texttt{"}) > 1) \tag{3.1}$$

It is important to note that the choice of a does not affect the decisions made at the second iteration: again, based on the restrictions imposed by Definition 1, the program cannot record the current character in the first iteration and transfer this information to the second iteration; hence (3.1) is well defined. When there is no corresponding a, let $Q_1(c) = \mathit{false}$.

We now define a family of predicates Q_i over the domain of characters $\mathcal{C}_0$ (including *null*) that describe the decision made at iteration i of the loop based on the current character, $c \in \mathcal{C}_0$. We use these predicates to show that the decisions taken at iteration i are always the same as those taken at iteration 1 (which is the *second* iteration), and depend solely on the current character. The definition is done by induction on i:

$$\begin{gathered} Q_{i+1}(c) \triangleq (\Delta_P(\texttt{"}\omega c\texttt{"}) > i+1) \\ \text{for some } \omega = a_0 \cdots a_i \text{ such that } \bigwedge_{j=0..i} Q_j(a_j) \end{gathered} \tag{3.2}$$

As before, the choice of ω is insignificant, and if no such ω exists, let $Q_{i+1}(c) = \mathit{false}$.

Claim 1. *$Q_i(c) = Q_1(c)$ for any $i \in \mathbb{N}^+$ and $c \in \mathcal{C}_0$.*

Proof. The definition of Q_i is based on the choice of P at iteration i when running on a string of length $i+1$. At that point, the situation is that $0 < i < \mathit{len}$. Therefore, again, the observations of P at iteration i are no different from its observations at iteration 1, assuming the same current character c; therefore $Q_i(c) = Q_1(c)$. □

The reason Claim 1 states that $Q_i(c) = Q_1(c)$ instead of $Q_i(c) = Q_0(c)$ is that according to our restrictions, the behaviour of P when operating on the first character of the string might differ from its behaviour on all other characters (this is because P can compare the iteration index to zero). Note that a similar issue does not occur concerning the last character of the string as the latter is always *null*.

Theorem 2 (Memoryless Squeeze)**.** *Let P be a memoryless forward loop. We construct a buffer* $\texttt{"}a\omega b\texttt{"}$ *where $a, b \in \mathcal{C}$ and $\omega \in \mathcal{C}^*$.*

1. *If $\Delta_P(\texttt{"}a\omega b\texttt{"}) = 1 + |\omega|$, then $\Delta_P(\texttt{"}ab\texttt{"}) = 1$.*

2. *If $\Delta_P(\texttt{"}a\omega b\texttt{"}) > 1 + |\omega|$, then $\Delta_P(\texttt{"}ab\texttt{"}) > 1$.*

Proof of Theorem 2. Let $a\omega b = a_0 a_1 \cdots a_{|\omega|+1}$ be the characters of $a\omega b$ (in particular, $a_0 = a$, $a_{|\omega|+1} = b$).

1. Assume $\Delta_P(\texttt{"}a\omega b\texttt{"}) = 1 + |\omega|$, then $Q_i(a_i)$ for all $0 \le i \le |\omega|$, and $\neg Q_{|\omega|+1}$. Therefore, $Q_0(a)$ (since $a_0 = a$), and $\neg Q_{|\omega|+1}(b)$. From Claim 1, also $\neg Q_1(b)$. Hence $[\![P]\!](\texttt{"}ab\texttt{"})$ completes the first iteration and exits the second iteration; so $\Delta_P(\texttt{"}ab\texttt{"}) = 1$.

2. Assume $\Delta_P(\texttt{"}a\omega b\texttt{"}) > 1 + |\omega|$, then $Q_i(a_i)$ for all $0 \le i \le |\omega| + 1$. In this case we get $Q_0(a)$ and $Q_{|\omega|+1}(b)$. Again from Claim 1, $Q_1(b)$. Hence $[\![P]\!](\texttt{"}ab\texttt{"})$ completes at least two iterations, and $\Delta_P(\texttt{"}ab\texttt{"}) > 1$. □

Theorem 3 (Memoryless Equivalence)**.** *Let F be a memoryless specification with forward traversal and character set X, and P a memoryless forward loop. If for every character sequence $\omega \in \mathcal{C}^*$ of length $|\omega| \le 2$ it holds that $[\![P]\!](\texttt{"}\omega\texttt{"}) = F(\texttt{"}\omega\texttt{"})$, then for any string buffer $s \in \mathcal{S}$ (of any length), $[\![P]\!](s) = F(s)$.*

Proof. Assume by contradiction that there exists a string $s \in \mathcal{S}$ on which P and F disagree, i.e., $[\![P]\!](s) \neq F(s)$. We show that we can construct a string s' such that $[\![P]\!](s') \neq F(s')$ and $|s'| \le 2$, which contradict our hypothesis.

We define $\Delta_F(s)$ as the number of iterations the specification F performs before returning. Definition 1 ensures that $0 \le \Delta_F(s)$ and $\Delta_F(s) \le \texttt{strlen}(s)$. By assumption, F is a forward loop, i.e., *start* $= 0$ and *end* $=$ *len*. Thus, $\Delta_F(s)$ is the length of the *longest prefix* τ of s such that $\tau \in \overline{X}^*$.

Since $[\![P]\!](s) \neq F(s)$, we know that $\Delta_P(s) \neq \Delta_F(s)$. If $\texttt{strlen}(s) \le 2$, we already have our small counterexample. Otherwise, we consider two cases.

<u>Case 1: $\Delta_P(s) < \Delta_F(s)$.</u> If $\Delta_P(s) = 0$, let $s' = \texttt{"}a\texttt{"}$ where a is the first character of s. According to Theorem 1, $\Delta_P(s') = 0$. However, $a \notin X$ (otherwise $\Delta_F(s) = 0 = \Delta_P(s)$, which we assumed is false), therefore $\Delta_F(s') = 1$. ∘

If $\Delta_P(s) > 0$, we decompose s into $\texttt{"}a\omega b\omega'\texttt{"}$, such that $\Delta_P(s) = |\omega| + 1$. We have $|a\omega b| > |\omega| + 1$, hence by Theorem 1, $\Delta_P(\texttt{"}a\omega b\texttt{"}) = \Delta_P(\texttt{"}a\omega b\omega'\texttt{"}) = |\omega| + 1$. Let $s' = \texttt{"}ab\texttt{"}$; by Theorem 2, we obtain $\Delta_P(s') = 1$. We know that $\Delta_F(\texttt{"}a\omega b\omega'\texttt{"}) \ge |\omega| + 2$, so $a\omega b \in X^*$, in particular $a, b \in X$. Therefore $\Delta_F(s') = 2$. ∘

Case 2: $\Delta_P(s) > \Delta_F(s)$. If $\Delta_F(s) = 0$, let $s' =$ "a" where a is the first character of s. Since $\Delta_P(s) \geq |s'| = 1$, and according to Theorem 1, we get $\Delta_P(s') \geq 1$. $\circ$

If $\Delta_F(s) > 0$, we again decompose s into "$a\omega b\omega'$", this time such that $\Delta_F(s) = |\omega| + 1$. From this construction we get $a\omega \in X^*$ (in particular $a \in X$) and $b \notin X$. Since $\Delta_P(s) \geq |\omega| + 2 = |a\omega b|$, and according to Theorem 1, we get $\Delta_P($"$a\omega b$"$) \geq |\omega| + 2 > |\omega| + 1$. Let $s' =$ "ab", and we know from Theorem 2 that $\Delta_P(s') > 1$. In contrast, from $a \in X, b \notin X$ established earlier, $\Delta_F(s') = 1$. $\circ$

In both cases (each with its two sub-cases, tagged with $\circ$), the end result is some $|s'| = \{1, 2\}$ for which $\Delta_P(s') \neq \Delta_F(s')$. This necessitates that $[\![P]\!](s') \neq F(s')$. $\square$

We note that it is easy to see that we can allow simple loops to start scanning the string from the nth character of the string instead of the first one provided we test that the program is memoryless for strings up to length of $n + 3$.

Unterminated Loops. Some library functions (*e.g.* `rawmemchr`) do not terminate on a null character, and even potentially perform unsafe reads. Still, we want to be able to replace loops in the program with such implementations as long as they agree on all safe executions and do not introduce new unsafe executions. This is done with a small adjustment to Definition 3 which allows for unsafe specifications. The details are mostly mundane and are therefore omitted.

3.4 Bounded Verification of Memorylessness

We implemented the bounded verification as an LLVMpass. The input LLVMbitcode was instrumented with `assert` commands that check the memorylessness conditions. The instrumented bitcode was then fed to KLEE, which verified no assertion violations occur on strings of length three and under. For example, whenever two integer values are compared, an instrumentation is inserted before the comparison to check that the values compared are either i and len or i and zero.

It is routine to make sure that this bounded verification is sufficient to show a loop is memoryless for all lengths provided that the program respects certain easily-check-able syntactic properties. These condition pertains to the way live variables are used as well as, effectively, that in every iteration a variable either increases its value by one or that its value is not changed in any iteration. We show that if the program presents such

Table 3.1: Loops remaining after each additional filter.

	Initial loops	Inner loops	Pointer calls	Array writes	Multiple ptr reads
bash	1085	944	438	264	45
diff	186	140	60	40	14
awk	608	502	210	105	17
git	2904	2598	725	495	108
grep	222	172	72	42	9
m4	328	286	126	78	12
make	334	262	129	102	13
make	207	172	88	67	20
sed	125	104	35	19	1
ssh	604	544	227	84	12
tar	492	432	155	106	33
libosip	100	95	39	30	25
wget	228	197	115	83	14
Total	7423	6448	2419	1515	323

a uniform behaviour in its first three iterations, it is bound to do so in any iteration. Thus, it suffices to check these properties on strings of length up to three.

Using our technique, we could prove that 85 loops out of the 115 meet the necessary conditions, spending on average less than three seconds per loop. Invalid loops typically contain constants other than zero, or change the read value by some constant offset (e.g., in `tolower` and `isdigit`).

3.5 Loop Database

We perform our evaluation on loops from 13open-source programs: *bash* , *diff*, *awk* , *git* , *grep*, *m4* , *make*, *make*, *sed*, *ssh*, *tar*, *libosip*and *wget*. These programs were chosen because they are widely-used and operate mostly on strings.

The process for extracting loops from these programs was semi-automatic. First, we used LLVMpasses to find 7,423 loops in these programs and filter them down to 323 candidate memoryless loops. Then we manually inspected each of these 323 loops and excluded the ones still not meeting all the criteria for memoryless loops. The next sections describe in detail these two steps.

3.5.1 Automatic Filtering

After compiling each of the programs to LLVMIR, we apply LLVM's *mem2reg* pass. This pass removes load and store instructions operating on local variables, and is needed in our next step. LLVM's *LoopAnalysis* was then used to iterate through all the loops in the program, and filter out loops which are not memoryless. We automatically prune loops that have inner loops and then loops with calls to functions that take pointers as arguments or return a pointer.

Then, we filter out loops containing writes into arrays. We assume that due to *mem2reg* pass, any remaining store instructions write into arrays and not into local variables. Therefore we miss loops where this assumption fails, as they get excluded based on containing a store instruction. Finally, we remove loops with reads from multiple pointer values. This ensures that we only keep loops with reads of the form $p_0 + i$ as per Definitions 1 and 2 of memoryless loops.

Table 3.1 shows, for each application considered, how many loops are initially selected (column *Initial loops*) and how many are left after each of the four filtering steps described above (e.g., column *Pointer calls* shows how many loops are left after both loops with inner loops and loops with calls taking or returning pointers are filtered out). In the end, we were left with between 9 and 108 loops per application, for a total of 323 loops.

3.5.2 Manual Filtering

We manually inspected the remaining 323 loops and manually excluded any loops that still did not meet the memoryless loops criteria from §3.1.

Two loops had `goto` in them, which meant they jumped to some other part of the function unrelated to the loops. Three loops had I/O side effects, such as outputting characters with `putc` (note that the automatic *pointer calls* filter removed most of the other I/O related loops).

A total of 74 loops did not return a pointer, and an additional 70 loops had a return statement in their body. 28 loops had too many arguments. For example, incrementing a pointer while it is smaller than another pointer would belong into this category, as the other pointer is an “argument” to the loop. Finally, 31 loops had more than one output, *e.g.* both a pointer and a length.

Note that some of these loops could belong into multiple categories, we just record the reason for which they were excluded during our manual inspection. In total, we manually excluded 208 loops, so we were left with 323 - 208 = 115 memoryless loops on which to apply our synthesis approach.

As part of this manual step, we also extracted each loop into a function with a `char*`

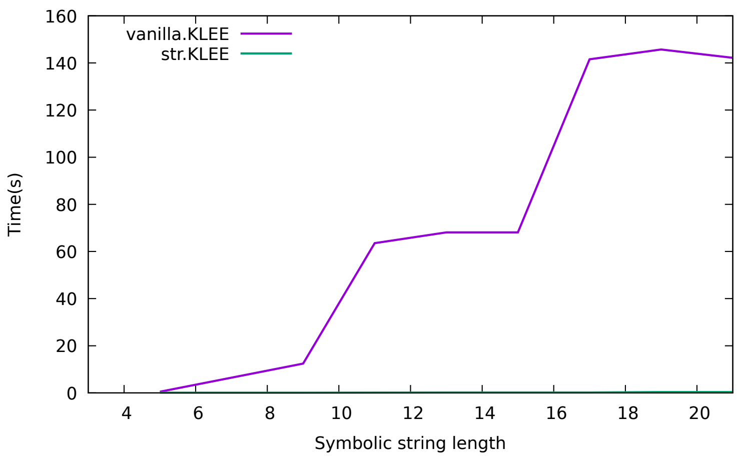


Figure 3.2: Mean time to execute all loops with str.KLEE and vanilla.KLEE, as we increase the length of input strings.

`loopFunction(char*)` signature. While this could be automated at the LLVMlevel, we felt it is important to be able to see the extracted loops at the source level.

3.6 Loop Summaries in Symbolic Execution

The next section discusses a scenario that can benefit from our loop summarisation approach: symbolic execution. Kapus et al. also explore the benefits of refactoring and compiler optimizations.

Recent years have seen the development of several efficient constraint solvers for the theory of strings, such as CVC4 [16], HAMPI [88] and Z3str [18, 153, 154]. These solvers can directly benefit symbolic execution, a program analysis technique that we also use in our approach. However, to be able to use these solvers, constraints have to be expressed in terms of the finite vocabulary that they support; standard string functions can be easily mapped to this vocabulary, but custom loops cannot.

To measure the benefits of using a string solver instead of directly executing the original loops, we wrote an extension to KLEE [27] (based on KLEE revision 9723acd) that can translate our loop summaries into constraints over the theory of strings and pass them to Z3 version 4.6.1. We refer to this extension as *str.KLEE*, and to the unmodified version of KLEE as *vanilla.KLEE*.

Figure 3.2 shows the average time difference across all loops between these two versions of KLEE when we grow the symbolic string length. We use a 240-second timeout and show the average execution time across all loops. For small strings of up

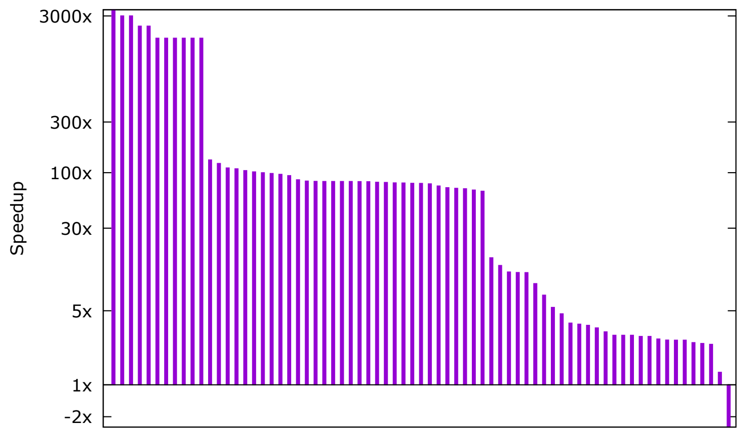


Figure 3.3: The speedup for each loop by str.KLEE over vanilla.KLEE for inputs of length 13, sorted by speedup value.

to length 8 the difference is negligible, but then it skyrockets until we hit a plateau where some loops start to time out with vanilla.KLEE. In contrast, str.KLEE's average execution time increases insignificantly, with an average execution time under 0.36s for all string lengths considered.

Figure 3.3 shows the speedup achieved for each loop by str.KLEE over vanilla.KLEE for symbolic input strings of length 13. For over half of the loops, str.KLEE achieves a speedup of more than two orders of magnitude, with several loops experiencing speedups of over 1000x (these include loops where str.KLEE times out, and where the actual speedup might be even higher). For others, we see smaller but still significant speedups. There is a single loop where str.KLEE does worse by a factor of 2.5x; this is a *strlen* function where it takes 1.4s, compared to 0.5s for vanilla.KLEE.

Our approach, as presented, is limited by the single pointer input/output interface to which loops have to conform. This restriction could be relaxed. For example, allowing an integer output instead of a pointer could be achieved with minor engineering effort. Allowing for loops that take two strings as input would be a larger effort. It would require both moderate engineering effort and a new small-model theorem. The synthesis will also require new gadgets conforming to the two-pointer interface. The new small-model theorem could be difficult to prove because the loop traverses two lists, but could exploit the fact that the loops are traversed in-sync.

We recognise that some of the loops we summarise could be recognised by more lightweight approaches such as source code pattern matching or scalar evolution approaches such as the one used by LLVM's LoopIdiomRecognize. However, it would be difficult to apply these approaches for complex loops that require additional modifications, such as conditionally incrementing a pointer after loop exit, setting it to the end of the string etc. More generally, pattern matching approaches require great manual effort in finding the patterns and then encoding them.

In §3.6, we show large increases in the scalability of symbolic execution with our summaries. This does not directly imply the same speed-ups would be observed when running whole programs with loops summaries, however we believe this work is an important step towards scaling symbolic execution to large strings.

Similar to our work, S-Looper [150] automatically summarises loops with the aim of improving program analysis. Their technique uses static analysis to enhance buffer-overflow detection. Our work is more general in that it is applicable to any analysis that operates on C directly, generating human-readable summaries that can even be used for refactoring.

Godefroid and Luchaup [56] use partial loop summarisation to enable concolic execution to reason about multiple paths through a loop at once. Their summaries consist of pre- and post-conditions, which they automatically infer during concolic execution. Similarly, *loop-extended symbolic-execution* [126] uses a combination of symbolic execution and static analysis to summarise loops in order to speed up symbolic execution. As for S-Looper, these two approaches are intertwined with their analysis, unlike our approach which can be immediately used in any technique.

STOKE [127] is an assembly level superoptimizer that speeds up loop-free code segments. With its recent extension to loops [31] their work is similar in spirit. They also use bounded verification to aid synthesis, but instead of a small-world theorem they use a sound verifier to generalise to arbitrary bounds. Their work focuses on optimising libc functions, whereas our work focuses on summarising loops in arbitrary programs, therefore we believe the work is complementary.

Srivastava et al. [136] present an approach synthesising loops from pre- and post-conditions using a verifier. While more precise, they require user-specified annotations, making it inapplicable as an automatic summarisation technique.

LLVM's LoopIdiomRecognize pass attempts to replace loops that match `memset` or `memcpy` patterns and is quite specific to these functions (other compilers, such as GCC, have similar passes that recognise patterns). It detects induction variables from which it can recognise stride load and store instructions. Their to-do includes functions like `strlen` for over 6 years, showing that such passes require significant expertise to implement. By contrast, our approach is more general and can easily be extended.

Program equivalence may be considered one of the most important problems in formal verification and has been the subject of decades of research [140]. Due to the vast literature on the topic and space, we only briefly review the subject.

Proving program equivalence is useful in many domains ranging from translation validation [93, 105, 113, 122, 130], regression verification [57, 58], automatic merging [135], semantic differencing [38], and cross-version verification [69, 95].

One common approach for attacking the problem, e.g., [148], is establishing a simulation invariant between the states of two programs. Tracking the simulation enables defining a so-called correlating semantics which allows reasoning about correlated (interleaved) execution of two programs [14, 38, 142]. In contrast to these techniques, our approach focuses on establishing the equivalence of programs without co-executing them, but instead examines their input/output behaviour on bounded examples using symbolic execution.

Symbolic execution-based methods [27, 30, 32, 33, 112, 117] often focus on practical equivalence verification up to a certain input bound. In contrast, we speculatively search for a synthesised program that agrees with the investigated loop on bounded inputs, and develop a small model theorem [115] which allows us to lift symbolic execution validated bounded equivalence to full equivalence.

Chapter 4

Size reduction for safety verification of Array manipulating programs

This chapter is based on the results published in [79].

Automatic verification of array manipulating programs is a challenging problem because it often amounts to the inference of inductive quantified loop invariants which, in some cases, may not even be first-order expressible. In this paper, we suggest a novel verification technique that is based on induction on user-defined *rank* of program states as an alternative to loop-invariants. Our technique, dubbed *inductive rank reduction*, works in two steps. Firstly, we simplify the verification problem and prove that the program is correct when the input state contains an input array of length ℓ_{B} or less, using the length of the array as the rank of the state. Secondly, we employ a *squeezing function* $\curlyvee$ which converts a program state m with an array of length $\ell > \ell_{\mathsf{B}}$ to a state $\curlyvee(m)$ containing an array of length $\ell - 1$ or less. We prove that when $\curlyvee$ satisfies certain natural conditions then if the program violates its specification on m then it does so also on $\curlyvee(m)$. The correctness of the program on inputs with arrays of arbitrary lengths follows by induction.

We make our technique automatic for array programs whose length of execution is proportional to the length of the input arrays by (i) performing the first step using symbolic execution, (ii) verifying the conditions required of $\curlyvee$ using Z3, and (iii) providing a heuristic procedure for synthesizing $\curlyvee$. We implemented our technique and applied it successfully to several interesting array-manipulating programs, including a bidirectional summation program whose loop invariant cannot be expressed in first-order logic while its specification is quantifier-free.

4.1 Introduction

Automatic verification of array manipulating programs is a challenging problem because it often amounts to the inference of inductive quantified loop invariants. These invariants are frequently quite hard to come up with, even for seemingly simple and innocuous program, both automatically and manually. The purpose of this work is to suggest an alternative kind of correctness witness, which is often simpler than inductive invariants and hence more amenable to automated search.

Loop invariants, the basis of traditional verification approaches, offer an induction scheme based on the time axis, *i.e.*, on the number of loop iterations. We suggest an alternative approach in which induction is carried out on the space axis, i.e. on a (user-defined notion of the) *rank* (e.g., size) of the program state. This is particularly useful in the setting of infinite-state systems, where the size of the state may be unbounded. In this induction scheme, establishing the induction step relies on a *squeezing function* $\curlyvee : \Sigma \to \Sigma$ (read $\curlyvee$ as *squeeze*) that maps program states to lower-ranked program states (up to a given minima). Roughly speaking, the squeezing function should satisfy the following conditions, described here intuitively and formalized in Definition 7:

- **Initial anchor.** $\curlyvee$ maps initial states to initial states.
- **Simulation inducing.** $\curlyvee$ induces a certain form of simulation between the program states and their squeezed counterparts.
- **Fault preservation.** $\curlyvee$ maps unsafe states to unsafe states.

Our main theorem (Theorem 4) shows that if these conditions are satisfied then P is correct, provided it is correct on its *base*, i.e., on the states with minimal rank. The crux of the proof is that as a consequence of the aforementioned conditions, if P violates its specification on a state m then it also violates it on $\curlyvee(m)$. Hence, if P satisfies the specification on the base states, by induction it satisfies it on any state.

The function $\curlyvee$ itself can be given by the user or, as we show in Section 4.4, automatically obtained for a class of array programs which iterate over their input arrays looking for a particular element (e.g., `strchr`) or aggregating their elements (e.g., `max`). In our experiments, we utilized automatically synthesized squeezing functions to verify natural specifications of several interesting array-manipulating programs, some of which are beyond the capabilities of existing automatic techniques. Arguably, the key benefit of our approach is that the squeezing functions are often rather simple, and thus finding them and establishing that they satisfy the required properties is an easier task than the inference of loop invariants. For example, in the next section we show a program whose loop invariant cannot be expressed in first order logic but can be proven correct using a squeezing function which is first-order expressible, in fact, the reasoning about the automatically synthesized squeezing function is quantifier free.

The last point to discuss is the verification of the program on states in the base of $\curlyvee$. Here, we apply standard verification techniques but to a simpler problem: we need to establish correctness only on the base, a rather small subset of the entire state space. For example, for the programs in our experiments it is possible to utilize symbolic execution to verify the correctness of the programs on all arrays of length three or less. This approach is effective because on the programs in our benchmarks, the bound on the length of the input arrays also determines a bound on the length of the execution. As this aspect of our technique is rather standard we do not discuss it any further.

Outline. The rest of the chapter is structured as follows: We first give an informal overview of our approach (Section 4.2) which is followed by a formal definition of our technique and a proof of its soundness (Section 4.3). We continue with a description of our heuristic procedure for synthesizing squeezing functions (Section 4.4) and a discussion about our implementation and experimental results (Section 4.5). We then review closely related work (Section 4.6) and conclude (Section 4.7).

4.2 Overview

In this section, we give a high-level view of our technique.

Running example. Program `sum_bidi`, shown in Figure 4.1, computes the sum of the input array `a` in two ways: One computation accumulates elements from left to right, and the other — from right to left (assuming that indexes grow to the right). Ignoring its dubious usefulness, `sum_bidi` possesses an intricate property: the variables

```
void sum_bidi(int a[], int n)
{
   int l = 0;
   int r = 0;
   for (int i = 0; i < n; i++)
   {
      l += a[i];
      r += a[n - i - 1];
   }
   assert(l == r);
}
```

$$I \mathrel{\widehat{=}} \big(l + \mathrm{sum}(a[i:n]) = r + \mathrm{sum}(a[0:n-i])\big)$$

$$\begin{aligned}&\mathrm{sum}(a[j:k]) \mathrel{\widehat{=}}\\ &\quad \textbf{if } j < k\\ &\qquad \textbf{then } a[j] + sum(a[j+1:k])\\ &\qquad \textbf{else } 0\end{aligned}$$

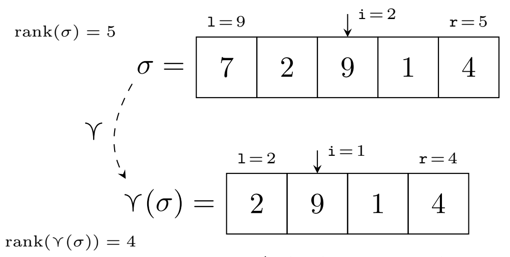
Figure 4.1: A bidirectional sum example and a loop invariant for it.

`l` and `r` are both computed to be the sum of the input array `a`. A natural property one expect to hold when the program terminates is that $l = r$.

The challenge. To verify the aforementioned postcondition when the length of the array is not known and *unbounded*, a loop invariant is often employed. It is important to remember, that a loop invariant must hold on all intermediate loop states — every time execution hits the loop header. For this reason, the loop invariant needed in this case is more involved than the mere assertion $l = r$ that follows the loop. The right side of Figure 4.1 shows a possible loop invariant for this scenario. Intuitively, the invariant says that `l` and `r` differ by the sum of the elements that they have not yet, respectively, accumulated. Notice that the invariant's formulation relies on a function $\mathrm{sum}(\cdot)$ for arrays (and array slices), the definition of which is also included in the figure. This definition is recursive; indeed, any definition of sum will require some form of recursion or loop due to the unbounded sizes of arrays in program memory. This kind of "logical escalation" (from quantifier-free $l = r$ to a fixed-point logic) makes such verification tasks challenging, since modern solvers are not particularly effective in the presence of quantifiers and recursive definitions.

Moreover, a system attempting to automate discovery of such loop invariants is prone to serious scalability issues since it has to discover the definition of $\mathrm{sum}(\cdot)$ along the way. The subject program `sum_bidi` effectively computes a sum, so this auxiliary definition is at the same scale of complexity as the program itself.

Our approach. We suggest to leverage the semantics already present in the subject program for a more compact proof of safety. Instead of having to summarize partial executions of the program via a loop invariant, we show that the program is correct for all arrays of size $0 \ldots r$ for some *base rank* r (the size of the array serves as the rank of



```
ϒ : {
      if (i > 0) {
         remove(a,0);
         i--;
         l -= a[0];
         r -= a[n - i];
      } else {
         remove(a,0);
      }
   }
```

Figure 4.2: A bidirectional sum example and its squeezing function.

the program state), and further show how to derive the correctness of the program for arrays of size $n > r$, from its correctness for arrays of size $n - 1$. To achieve the latter, we rely on a function that "squeezes" states in which the array length is n to states in which the array length is $n - 1$, as we illustrate next.

Continuing with the example `sum_bidi` described above, we use the function $\Upsilon : \Sigma \to \Sigma$, defined as a code block on the right side of Figure 4.2, to "squeeze" program states. In this case, the state consists of the variables ⟨`a`, `n`, `i`, `l`, `r`⟩, and it is squeezed by removing the first element of `a` and adjusting the indices and sums accordingly. The base rank here is $r = 0$, since any non-empty array can be squeezed in this manner. The bottom part of Figure 4.3 shows the effect of applying Υ to each of the states in the execution trace of `sum_bidi` on the example input `[7,2,9,1,4]`. The first property that is demonstrated by the diagram is the "initial anchor" property, stating that initial states are "squeezed" into initial states. As is obvious from the diagram, the execution on the squeezed array `[2,9,1,4]` is accordingly shorter, so Υ cannot be injective — in this case, $\Upsilon(\sigma_0) = \Upsilon(\sigma_1) = \sigma'_0$. Still, the sequence $\sigma'_0 \to \sigma'_1 \to \sigma'_2 \to \sigma'_3 \to \sigma'_4$ constitutes a valid trace of `sum_bidi`. This is the second property required of Υ, which we refer to as *simulation inducing* and define it formally in the next section.

Now, draw attention to *fault preservation*, the third property required of Υ: whenever a state σ falsifies the safety property φ, denoted $\sigma \not\models \varphi$, it is also the case that $\Upsilon(\sigma)$ falsifies the safety property, i.e. $\Upsilon(\sigma) \not\models \varphi$. In our example, the safety property can be formalized as $\varphi \mathrel{\widehat{=}} (i = n \to l = r)$. The reasoning establishing fault preservation is not immediate but still quite simple: if $\sigma \not\models \varphi$, it means that $i = n$ but $l \neq r$ (at σ). In that case, $a[n - i] = a[0]$; so $l' = l - a[0] \neq r - a[n - i] = r'$, where l', r' are the values of `l` and `r`, respectively, at state $\Upsilon(\sigma)$. Since i and n are both decremented[1] we get $\Upsilon(\sigma) \not\models \varphi$.

In this manner, from *the assumption that* $\Upsilon(\sigma_j)$*, for* $j = 0..5$*, induces a safe trace,*

[1]Notice that we assume a positive size ($n > 0$), otherwise the array cannot be squeezed in the first place.

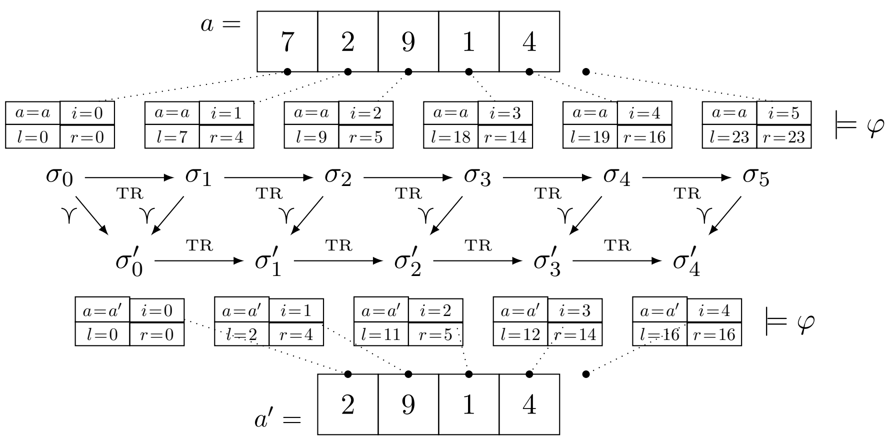


Figure 4.3: Example trace of `sum_bidi`, and the corresponding shrunken image.

we conclude that σ_j is safe as well. This lends the notion of constructing a proof by induction on the size of the initial state σ_0, provided that Υ cannot "squeeze forever" and that we can verify all the minimal cases more easily, *e.g.* with bounded verification. This is definitely true for `sum_bidi`, since the minimal case would be an empty array, in which the loop is never entered. In some situations the minima contains states with small but not empty arrays. In general, if one can verify that the program is correct when started with a minimal initial state, thus establishing the base case of the induction, our technique would lift this proof to hold for unbounded initial states. In particular, if the length of the program's execution trace can be bounded based on the size of the initial state then bounded model checking and symbolic execution can be lifted to obtain unbounded correctness guarantee.

It is worth mentioning at this point that Υ is in no sense "aware" that it is, in fact, reasoning about sums. It only has to handle scalar operations, in this case subtraction (as the counterpart of addition that occurs in `sum_bidi`; the same will be true for any other commutative, invertible operation.) The folding semantics arises spontaneously from the induction over the size of the array.

Recap. We suggest a novel verification technique that is based on induction on the size of the input states as an alternative to loop-invariants. The technique is based on utilizing a squeezing function which converts high-ranked states into low-ranked ones, and then applying a standard verification technique to establish the correctness of the program on the minimally-ranked states. In a manner analogous to that which is carried out with "normal" verification using loop invariants, the squeezer has to uphold the three properties described in Section 4.1, namely *initial anchor*, *simulation inducing*, and *fault preservation*. (See Section 4.3 for a formal definition.) These properties ensure that the mapping induces a valid reduction between the safety of any trace and that of its squeezed counterpart.

Why bother. The attentive readers may ask themselves, given that both loop invariants and squeezers incur some proof obligations for them to be employed for verification, what benefit may come of favoring the latter over the former. While the verification condition scheme proposed here is not inherently simpler (and arguably less so) than its Floyd-Hoare counterpart, we would like to point out that the *squeezer itself*, at least in the case of `sum_bidi`, *is indeed simpler* than the loop invariant that was needed to verify the same specification. It is simpler in a sense that it resides in a *weaker logical fragment*: while the invariant relies on having a definition of (partial) sums, itself a recursive definition, the squeezer $\curlyvee$ can be axiomatized in a quantified-free formula using a theory of strings (sequences) [19] and linear arithmetic. In Section 4.4 we take advantage of the simplicity if the squeezing function, and show that it is feasible to *generate it automatically* using a simple enumerative synthesis procedure.

On top of that, it is quite immediate to see that the induction scheme outlined above is still sound even if the properties of $\curlyvee$ (initial anchor, simulation, and fault preservation) only hold for *reachable* states. Obviously, the set of reachable states cannot be expressed directly — otherwise we would have just used its axiomatization together with the desired safety property, making any use of induction superfluous. Even so, if we can acquire any known property of reachable states, *e.g.* through a preliminary phase of abstract interpretation [35], then this property can be added as an assumption, simplifying $\curlyvee$ itself. A keen reader may have noticed that the specification of `sum_bidi` has been written down as $\varphi \mathrel{\widehat{=}} (i = n \rightarrow l = r)$, while a completely honest translation of the assertion would in fact produce a slightly stronger form, $\varphi' \mathrel{\widehat{=}} (i \geq n \rightarrow l = r)$. This was done for presentation purposes; in an actual scenario the "proper" specification φ' is used, and a premise $0 \leq i \leq n$ is assumed. Such range properties are prevalent in programs with arrays and indexes, and can be discovered easily using static analysis, e.g., using the Octagon domain [102].

This final point is encouraging because it gives rise to a hybrid approach, where a *partial* loop invariant is used as a baseline — verified via standard techniques — and is then *stengthened* to the desired safety property via squeezer-based verification. Or, the order could be reversed. There can even be alternating strengthening phases each using a different method. These extended scenarios are potentialities only and are matter for future work.

4.3 Verification by Induction over State Size

In this section we formalize our approach for verifying programs that operate over states (inputs) with an unbounded size. The approach mimics induction over the state size. The base case of the induction is discharged by verifying the program for executions over "small" low-ranked states (to be formalized later). For the induction step, we need to deduce correctness of executions over "larger" higher-ranked states from the correctness of executions over "smaller" states. This is facilitated by the use of a *simulation-inducing squeezing function* $\curlyvee$. Intuitively, the function transforms a state m into a corresponding "smaller" state $\curlyvee(m)$ such that executions starting from the latter simulate executions starting from the former. The simulation ensures that correctness of the executions starting from the smaller state, $\curlyvee(m)$, implies correctness of the executions starting from the larger one, m.

Transition systems and safety properties. To formalize our technique, we first define the semantics of programs using *transition systems.* The is quite standard.

Definition 5 (Transition Systems)**.** *A* transition system $TS = (\Sigma, Init, Tr, P)$ *is a quadruple comprised of a* universe *(a set of states)* Σ*, a set of* initial states $Init \subseteq \Sigma$*, a* transition relation $Tr \subseteq \Sigma \times \Sigma$*, and a set of* good states $P \subseteq \Sigma$*.*

A *trace* of TS is a (finite or infinite) sequence of states $\tau = \sigma_0, \sigma_1, \ldots$ such that for every $0 \leq i < |\tau|$, $(\sigma_i, \sigma_{i+1}) \in Tr$. In the following, we write Tr^k, for $k \geq 0$ to denote k self compositions of Tr, where $Tr^0 = Id$ denotes the identity relation. That is, $(\sigma, \sigma') \in Tr^k$ if and only if σ' is reachable from σ by a trace of length k (where the length of a trace is defined to be the number of transitions along the trace).

A transition system $TS = (\Sigma, Init, Tr, P)$ is *safe* if all its *reachable states* are good (or "safe"), where the set of reachable states is defined, as usual, to be the set of all states that reside on traces that start from the initial states. A *counterexample trace* is a trace that starts from an initial state and includes a "bad" state, i.e., a state that is not in P. The transition system is safe if and only if it has no counterexample traces.

Simulation-inducing squeezer. To present our technique, we start by formalizing the notion of a simulation-inducing squeezing function (*squeezer* for short).

Definition 6 (Squeezing function)**.** *Let X be a set and $\preceq$ a well-founded partial order over X. Let $B \supseteq \min(X)$ be a* base *for X, where $\min(X)$ is the set of all the minimal elements of X w.r.t. $\preceq$, and let $\rho : \Sigma \to X$ be a* rank *on the program states. A function $\curlyvee : \Sigma \to \Sigma$ is a* squeezing function*, or* squeezer *for short,* with base *B if for every state $\sigma \in \Sigma$ such that $\rho(\sigma) \in X \setminus B$, it holds that $\rho(\curlyvee(\sigma)) \prec \rho(\sigma)$.*

That is, $\curlyvee$ must strictly decrease the rank of any state unless its rank is in the base, B. We refer to states whose size is in B as *base states*, and denote them $\Sigma_B = \{\sigma \in \Sigma \mid \rho(\sigma) \in B\}$. We denote by $\Sigma_{\overline{B}} = \Sigma \setminus \Sigma_B$ the remaining states. Since $\preceq$ is well-founded and all the minimal elements of X w.r.t. $\preceq$ must be in B (additional elements may be included as well), any maximal strictly decreasing sequence of elements from X will reach B (i.e., will include at least one element from B). Hence, the requirement of a squeezer ensures that any state will be transformed into a base state by a *finite* number of $\curlyvee$ applications.

Example 2. *In our examples, we use $(\mathbb{N}, \leq)$ as a well-founded set, and define the base as an interval $[0, k]$ for some (small) $k \geq 0$. While it suffices to define $B = \min(\mathbb{N}) = \{0\}$, it is sometimes beneficial to extend the base to an interval since it excludes additional states from the squeezing requirement of $\curlyvee$ (see Section 4.5). For array-manipulating programs, the rank used is often (but not necessarily) the size of the underlying array, in which case, the "squeezing" requirement is that whenever the array size is greater than k, the squeezer must remove at least one element from the array. For example, for `sum_bidi` (Figure 4.2), we consider $k = 0$, i.e., the base consists of arrays of size 0, and, indeed, whenever the array size is greater than 0, it is decremented by $\curlyvee$. For arrays of size 0, $\curlyvee$ behaves as the identity function (this case is omitted from the figure). In addition, whenever the state contains more than one array, we will use the sum of lengts of all arrays as a rank.*

Definition 7 (Simulation-inducing squeezer)**.** *Given a transition system $TS = (\Sigma, Init, Tr, P)$, a squeezer $\curlyvee : \Sigma \to \Sigma$ is* simulation-inducing *if the following three conditions hold for every $\sigma \in \Sigma$:*

- **Initial anchor:** *if $\sigma \in Init$ then $\curlyvee(\sigma) \in Init$ as well.*

- **Simulation inducing:** *there exist $n_\sigma \geq 1$ and $m_\sigma \geq 0$ such that if $(\sigma, \sigma') \in Tr^{n_\sigma}$ then $(\curlyvee(\sigma), \curlyvee(\sigma')) \in Tr^{m_\sigma}$, i.e., if σ reaches σ' in n_σ steps, then the same holds for their $\curlyvee$-images, except that the number of steps may be different.*

- **Fault preservation:** *if $\sigma \notin P$ then $\curlyvee(\sigma) \notin P$ as well.*

The definition implies that $\{(\sigma, \curlyvee(\sigma)) \mid \sigma \in \Sigma\}$ is a form of a "skipping" simulation relation, where steps taken both from the simulated state, σ, and from the simulating state, $\curlyvee(\sigma)$, may skip over some states. This allows the simulated and the simulating execution to proceed in a different pace, but still remain synchronized. In fact, to ensure that we obtain a "skipping" simulation, it suffices to consider a weaker simulation inducing requirement where the parameter m_σ that determines the number of steps in the simulating trace depends not only on σ but also on σ' and may be different for each σ'. Note that for deterministic programs (as we use in our experiments) these requirements are equivalent. Another possible, yet stronger, relaxation is to weaken the requirement that $(\curlyvee(\sigma), \curlyvee(\sigma')) \in Tr^{m_\sigma}$ into $(\curlyvee(\sigma), \curlyvee(\sigma')) \in Tr^{i}$ for some $0 \leq i \leq m_\sigma$.

Example 3. *To illustrate the simulation inducing requirement, recall the program `sum_bidi` from Example 2. For the base states ($n = 0$), $\curlyvee$ behaves as the identity function. Hence, for such states the skipping parameters n_σ and m_σ are both 1 (letting each step be simulated by itself). For non-base states, n_σ, the "skipping" parameter of σ, is still 1, while m_σ, the "skipping" parameter of $\curlyvee(\sigma)$, is 0 if σ is an initial state, and 1 otherwise. This accounts for the fact that $\curlyvee$ truncates the head of the array; hence, the first step in an execution is skipped in the corresponding "squeezed" execution, while the rest of the steps are synchronized in both executions (see Figure 4.3 for an illustration).*

Intuitively, one may conjecture that given a loop that iterates over an array, it will essentially perform fewer iterations when run on $\curlyvee(\sigma)$ than it does on σ, always resulting in $m_\sigma \leq n_\sigma$. The following example shows that this is not necessarily the case.

Example 4. *The program `is_sorted` (Figure 4.4) checks whether the input array elements are ascending by comparing all consecutive pairs. Our squeezer (for $n > 3$) checks whether the last three elements form an ascending sequence; if so, removes the last element, otherwise it removes the forth element from the right. Consider the*

```
bool is_sorted(int a[],int n) {
  for (int i=1;i<n;i++)
    if (a[i] < a[i-1])
      return false;
  return true;
}
```

```
ϒ:
   if (a[n-3] <= a[n-2] &&
       a[n-2] <= a[n-1])
        remove(a,n-1);
   else
       remove(a,n-4);
```

Figure 4.4: Another program with Υ demonstrating a scenario where $n_\sigma < m_\sigma$.

$$\begin{tikzcd}
\sigma_0 \arrow[r, dashed, "Tr^{n_{\sigma_0}}"] \arrow[d, "\Upsilon"'] & \sigma_1 \arrow[r, dashed, "Tr^{n_{\sigma_1}}"] \arrow[d, "\Upsilon"'] & \sigma_2 \arrow[r, dashed, "Tr^{n_{\sigma_2}}"] \arrow[d, "\Upsilon"'] & \cdots \arrow[r, dashed, "Tr^{n_{\sigma_{k-1}}}"] & \sigma_k \arrow[r, dashed, "Tr^{n_{\sigma_k}}"] \arrow[d, "\Upsilon"'] & \cdots & \models \varphi \\
\sigma'_0 \arrow[r, dashed, "Tr^{m_{\sigma_0}}"] \arrow[d, dashed, "\Upsilon"'] & \sigma'_1 \arrow[r, dashed, "Tr^{m_{\sigma_1}}"] \arrow[d, dashed, "\Upsilon"'] & \sigma'_2 \arrow[r, dashed, "Tr^{m_{\sigma_2}}"] \arrow[d, dashed, "\Upsilon"'] & \cdots \arrow[r, dashed, "Tr^{m_{\sigma_{j-1}}}"] & \sigma'_j \arrow[r, dashed, "Tr^{m_{\sigma_j}}"] & \cdots & \models \varphi \arrow[u, "\Uparrow"'] \\
\vdots \arrow[d, dashed, "\Upsilon"'] & \vdots \arrow[d, dashed, "\Upsilon"'] & \vdots \arrow[d, dashed, "\Upsilon"'] & \vdots \arrow[d, dashed, "\Upsilon"'] & & & \\
\sigma^\dagger_0 \arrow[r, dashed, "Tr^{m^\dagger_0}"] \arrow[d, phantom, "\in \Sigma_B"] & \sigma^\dagger_1 \arrow[r, dashed, "Tr^{m^\dagger_1}"] & \sigma^\dagger_2 \arrow[r, dashed, "Tr^{m^\dagger_2}"] & \cdots \arrow[r, dashed, "Tr^{m^\dagger_{r-1}}"] & \sigma^\dagger_r \arrow[r, dashed, "Tr^{m^\dagger_r}"] & \cdots & \models \varphi \arrow[uu, dashed, "\Uparrow"'] \\
 & & & & & &
\end{tikzcd}$$

Figure 4.5: Soundness proof sketch; an arbitrary trace can be reduced to a low-ranked trace by countable applications of Υ. Since ranks form a well-founded set, a base element is encountered after finitely many such reductions. Arrows with vertical ellipses indicate alternating applications of Υ and Tr^*, except for initial states where Definition 7(1) ensures straight applications of Υ alone.

input a=1,0,2,3,1 and the squeezed a'=1,2,3,1. `is_sorted(a)` *terminates after one iteration, but* `is_sorted(a')` *after three iterations. Let $\sigma = [a, i \mapsto 1]$. The simulation inducing requirement can only be satisfied with $n_\sigma = 1$ and $m_\sigma = 3$. Since $Tr^{n_\sigma}(\sigma) = [a, ret = false]$, no smaller value of m_σ can satisfy the requirement that $Tr^{m_\sigma}\big(\Upsilon(\sigma)\big) = \Upsilon\big(Tr^{n_\sigma}(\sigma)\big)$.*

Checking if a squeezer is simulation-inducing. The initial anchor and fault preservation requirements are simple to check. To facilitate checking the simulation inducing requirement, we do not allow arbitrarily large numbers n_σ, m_σ but, rather, determine a bound N on the value of n_σ and a bound M on the value of m_σ. This makes the simulation inducing requirement stronger than required for soundness, but avoids the need to reason about pairs of states that are reachable by traces of unbounded lengths (n_σ and m_σ).

Using simulation-inducing squeezer for safety verification. Roughly, the existence of a simulation-inducing squeezer ensures that any counterexample to safety, i.e., an execution starting from an initial state and ending in a *bad* state (a state that falsifies the safety property), can be "squeezed" into a counterexample that starts from a "smaller" initial state. In this sense, the squeezer establishes the induction step for proving safety by induction over the state rank. To ensure the correctness of this argument, we need to require that a "bad" state may not be "skipped" by the simulation induced by the squeezer.

Formally, this is captured by the following definition.

Definition 8. *A transition system $TS = (\Sigma, \mathit{Init}, \mathit{Tr}, P)$ is* recidivist *if no "bad" state is a dead-end, i.e., $\sigma \notin P \implies \exists\sigma'.\,(\sigma, \sigma') \in \mathit{Tr}$, and that transitions leaving "bad" states lead to "bad" states, i.e., $\sigma \notin P \wedge (\sigma, \sigma') \in \mathit{Tr} \implies \sigma' \notin P$.*

Recidivism can be obtained by removing any outgoing transition of a bad state and adding a self loop instead. Importantly, this transformation does not affect the safety of the underlying program. In our examples, terminal states of the program are treated as self loops, thus ensuring recidivism.

Lemma 4. *Let $\curlyvee : \Sigma \to \Sigma$ be a simulation-inducing squeezer for a recidivist transition system $TS = (\Sigma, \mathit{Init}, \mathit{Tr}, P)$. For every $\sigma_0 \in \Sigma$, if there exists a counterexample that starts from σ_0, then there also exists a counterexample that starts from $\curlyvee(\sigma_0)$.*

The proof is constructive: given a counterexample trace from σ_0, we use the simulation-inducing parameters n_σ of the states σ along the trace to divide it into segments such that the first and last state of each segment are the ones used as synchronization points for the simulation and the inner ones are the ones "skipped" over. We then match each segment (σ, σ') with the corresponding trace of length m_σ from $\curlyvee(\sigma)$ to $\curlyvee(\sigma')$, whose existence is guaranteed by the simulation inducing requirement. The concatenation of these traces forms a counterexample trace from $\curlyvee(\sigma_0)$. Formally:

Proof. Let $\tau = \sigma_0, \sigma_1, \ldots, \sigma_n$ be a counterexample trace starting from an initial state $\sigma_0 \in \mathit{Init}$. If the counterexample is of length 0, then $\curlyvee(\sigma_0)$ is also a counterexample of length 0 (by the initial anchor and fault preservation requirements). Consider a counterexample of length $n > 0$. We show how to construct a corresponding counterexample from $\curlyvee(\sigma_0)$. We first split the indices $0, \ldots, n$ into (overlapping) intervals $I_0, \ldots, I_k$, where $I_0 = 0, \ldots, n_\sigma$, and for every $i \geq 1$, if the last index in I_{i-1} is j for $j < n$, then $I_i = j, \ldots, j + n_{\sigma_j}$. If $j + n_{\sigma_j} \geq n$, then $k := i$. Since *TS* is recidivist, we may assume, without loss of generality, that $j + n_{\sigma_j} = n$ (otherwise, because *TS* is recidivist and $\sigma_n \notin P$, we can exploit one of the transitions leaving σ_n, which necessarily exists and leads to a bad state, to extend the counterexample trace as needed.) We denote by $\mathit{first}(I_i)$, respectively $\mathit{last}(I_i)$, the smallest, respectively largest, index in I_i. By the definition of the intervals, for every $0 \leq i \leq k$, we have that $\mathit{last}(I_i) = \mathit{first}(I_i) + n_{\sigma_{first(I_i)}}$. Hence, the simulation inducing requirement for $\sigma_{\mathit{first}(I_i)}$ ensures that there exists a trace of $m_{\sigma_{\mathit{first}(I_i)}}$ steps from $\curlyvee(\sigma_{\mathit{first}(I_i)})$ to $\curlyvee(\sigma_{\mathit{last}(I_i)})$. Since $\sigma_{\mathit{first}(I_0)} = \sigma_0$ and for every $0 < i \leq k$, $\sigma_{\mathit{first}(I_i)} = \sigma_{\mathit{last}(I_{i-1})}$, we can glue these traces together to obtain a trace from $\curlyvee(\sigma_0)$ to $\curlyvee(\sigma_{\mathit{last}(I_i)})$. Finally, it remains to show that $\curlyvee(\sigma_{\mathit{last}(I_k)}) \notin P$. This follows from the fault preservation requirement, since $\mathit{last}(I_k) = n$, hence $\sigma_{\mathit{last}(I_k)} = \sigma_n \notin P$. □ □

Ultimately, the existence of a simulation-inducing squeezer implies that a counterexample can be "squeezed" to one that starts from a base initial state. Hence, to establish that the transition system is safe, it suffices to check that it is safe when the initial states are restricted to the base states, i.e., to $\mathit{Init} \cap \Sigma_B$.

Theorem 4 (Soundness)**.** *Let $\curlyvee : \Sigma \to \Sigma$ be a simulation-inducing squeezer with base B for a recidivist transition system $TS = (\Sigma, \mathit{Init}, \mathit{Tr}, P)$. If $TS_B = (\Sigma, \mathit{Init} \cap \Sigma_B, \mathit{Tr}, P)$ is safe then TS is safe.*

Proof. Suppose for the sake of contradiction, that $\{\sigma_i\}_{i=0}^d$ is a counterexample trace with minimal rank for σ_0 (such a state with a minimal rank exists since $\preceq$ is well-founded). Since TS_B is safe, it must be that $\sigma_0 \in \Sigma_{\overline{B}}$ (since $\sigma_0 \in \mathit{Init}$, while safety of TS_B ensures that no counterexample trace can start from $\mathit{Init} \cap \Sigma_B$). By Lemma 4, we have that $\curlyvee(\sigma_0)$ also has an outgoing counterexample trace. However, since $\sigma_0 \in \Sigma_{\overline{B}}$, we get that $\rho(\curlyvee(\sigma_0)) \prec \rho(\sigma_0)$, in contradiction to the minimality of σ_0. □ □

In all of our examples, the transitions of *TS* do not increase the rank of the state. In such cases, we can also restrict the state space of TS_B (and accordingly *Tr*) to the base states in Σ_B. Furthermore, in these examples, the size of the state (array) also determines the length of the executions up to a terminal state. Hence, bounded model checking suffices to determine (unbounded) safety of TS_B, and together with $\curlyvee$, also of *TS*.

As evident from the proof of Theorem 4, it suffices to require that $\curlyvee$ decreases the rank of the *initial* non-base states, and not of all the non-base states.

4.4 Synthesizing Squeezing Functions

So far we have assumed that the squeezer $\curlyvee$ is readily available, in much the same way that loop invariants are available — typically, as user annotations — in standard unbounded loop verification. As demonstrated by the examples in Sections 4.2 and 4.3, $\curlyvee$ is specific to a given program and safety property. Thus, it might be tedious to provide a different squeezer every time we wish to check a different safety property. In this section we show how to lighten the burden on the user by automating the process of obtaining squeezing functions for a class of typical programs that loop over arrays.

The solution for the *squeezer-inference* problem we take in this paper is to utilize a rather standard enumerative synthesis technique of multi-phase generate-and-test: We take advantage of the relative simplicity of $\curlyvee$ and provide a synthesis loop where we generate grammatically-correct squeezing functions and test whether they induce simulation.

4.4.1 Generate

First we note that while $\curlyvee$ is applied to arbitrary states in Definition 7, it is only required to reduce the rank of non-base states $\sigma \in B$. For states $\sigma \in B$ it is trivial to satisfy all the requirements by defining $\curlyvee(\sigma) = \sigma$. In the sequel, we therefore only consider squeezing functions whose restriction to B is the identity, and synthesize code for squeezing non-base states.

A central insight is that squeezing functions $\curlyvee$ for different programs still have some structure in common: for programs with arrays, squeezing amounts to removing an element from the array, and adjusting the index variables accordingly. Some more detailed treatment may be needed for general purpose variables, such as the accumulators `l` and `r` of `sum_bidi` (recall Figure 4.1), but the resulting expressions are still small.

We have found that, for the set of programs used in our experiments, $\curlyvee$ can be characterised by the grammar in Figure 4.6. The grammar allows for functions comprised of a single if statement, where in each branch an array is squeezed using the `remove` function, and several integer variables are set. Conditions are generated by composing

$$
\begin{array}{lcl}
\text{body} & ::= & \texttt{if}\ (\ \text{cond}\ ) \\
 & & \quad \texttt{remove(}\ \text{arr},\ \text{exp}_{index}\ \texttt{)}\ \left[\text{var}_{int}\ \texttt{=}\ \text{exp}_{int}\right]^{*} \\
 & & \texttt{else} \\
 & & \quad \texttt{remove(}\ \text{arr},\ \text{exp}_{index}\ \texttt{)}\ \left[\text{var}_{int}\ \texttt{=}\ \text{exp}_{int}\right]^{*} \\
\text{cond} & ::= & \text{elem}_{\tau}\ \diamond(\text{elem}_{\tau} \mid \text{const}_{\tau}) \\
 & \mid & \text{var}_{index}\ \diamond(\text{var}_{index} \mid \text{const}_{index}) \qquad\qquad \diamond\ ::=\ \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \\
 & \mid & \text{cond}\ \texttt{\&\&}\ \text{cond} \quad \mid \quad \text{cond}\ \texttt{||}\ \text{cond} \\
\textit{exp}\text{index} & ::= & \text{const}_{index} \mid \text{var}_{index} \mid \texttt{len}(\text{arr})\ \texttt{-}\ (\text{const}_{index} \mid \text{var}_{index}) \\
\text{exp}_{int} & ::= & \text{var}_{int}\ (\texttt{+} \mid \texttt{-})\ \text{elem}_{int} \\
\text{elem}_{\tau} & ::= & \text{arr}\ \texttt{[}\ \text{exp}_{index}\ \texttt{]} \\
\text{const}_{index} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \\
\text{const}_{\tau} & ::= & \texttt{0} \mid \text{other constants occurring in the program} \\
\multicolumn{3}{l}{\text{arr},\ \text{var}_{index},\ \text{var}_{int},\ \text{var}_{char}\ \text{— identifiers occurring in the program}}
\end{array}
$$

Figure 4.6: Program space for syntax-guided synthesis of Υ. Expressions are split into three categories: *index*, *int*, and *char* as described in Section 4.4. $\tau \in \{int, char\}$.

array elements, local variables and a fixed set of constants based on the given program, with standard comparison operators and boolean connectives. The semantics of `remove`(arr, position) are such that a single element is removed from the array at the specified position, and all index variables are adjusted by decrementing them if they are larger from the index of the element being removed. This behavior is hard-coded and is specific to array-based loops. Our experience has shown that a single conditional statement is indeed sufficient to cover many different cases (see Section 4.5).

To bound the search space, expressions and conditions have bounded sizes (in terms of AST height) imposed by the generator and the user selects the set of basic predicates from which the condition of the `if` statement is constructed. The resulting space, however, is still often too large to be explored efficiently. To reduce it, some type-directed pruning is carried out so that only valid functions are passed to the checker. Moreover, our synthesis procedure distinguishes between variables that are used as indices to the array (var_{index}) and regular integer variables (var_{int}), and does not mix between them. We further assume that we can determine, from analyzing the program's source code, which index variable is used with which array(s). So when generating expressions of the form arr`[` i `]` *etc.*, only relevant index variables are used. Also, we note that generated squeezers preserve in bounds access by construction.

4.4.2 Test

The *test* step checks whether a candidate squeezer that is generated by the synthesizer satisfies the requirements of Definition 7. For the simulation-inducing requirement, we restrict $n_m = 1..2$ and $m_m = 0..1$. The step is divided into three phases. In the first phase, candidates are checked against a bank of concrete program states (both reachable and unreachable). In the second phase, candidates are verified for a bounded array size, but with no restrictions on the values of the elements. Those that pass bounded verification enter the third phase where full, unbounded verification is performed.

The second and third phases of the test step require the use of an SMT solver. The second phase is useful since incorrect candidates may cause the solver to diverge when queried for arbitrary array sizes. Limiting the array size to a small number (we used 6) enables to rule out these candidates in under a second. To simplify the satisfiability checks, we found it beneficial to decompose the verification task. To do so, we take advantage of the structure of the squeezer, and split each satisfiability query (that corresponds to one of the requirements in Definition 7) into two queries, where in each query we make a different assumption regarding the branch the squeezer function takes. We note in this context that the capabilities of the underlying solver direct (or limit in some sense) the expressive power of the squeezer. In this aspect, it is also worth mentioning that sequence theory support for element removal helped to define squeezers format.

For the simulation inducing check, we further exploit the property that for the kind of programs and squeezers we consider, the transitions of the program usually do not change the truth value of the condition of the if statement in the definition of the squeezer. Namely, if m makes a transition to m' then either both of them satisfy the condition or both of them falsify it; either way, their definition of $\curlyvee$ follows the same branch. This form of preservation can be checked automatically using additional queries. When it holds, we can consider the same branch of the squeezer program in both the pre- and post-states, thus simplifying the query for checking simulation. Similarly, we can opportunistically split the transition relation of the program into branches (e.g., one that executes an iteration of the loop and one that exits the loop). In most cases, the same branch that was taken for m is also the one that needs to be taken from $\curlyvee(m)$ to establish simulation. This leads to another simplification of the queries, which is sound (i.e., never concludes that the simulation-inducing requirement holds when it does not), but potentially incomplete. We can therefore use it as a "cheaper" check and resort to the full check if it fails.

4.4.3 Filtering out unreachable states

For soundness, a squeezer needs to satisfy Definition 7 only on the reachable states. As we do not have a description of this set, for otherwise the verification task would be essentially voided, we need to ensure that the requirements of simulation-inducement on a safe over-approximation of this set. A simple over-approximation would be the set of all states. However, this over-approximation might be too coarse, indeed we noticed in our experiments that in some cases, unreachable states have caused phases 1, 2 and 3 to produce false negatives,*i.e.* , disqualify squeezers which can be used safely to verify the program. Therefore we used an over-approximation of reachable states using

1. Bound constraints on the index variables: the index is expected to be within bounds of the traversed array. This property can be easily verified using other verifiers or by applying our verifier in stages, first proving this property and then proving the actual specification of the verified procedure under the assumption that the property hold.

2. 2-step bounded reachability: We found out that for our examples, looking only at states that are reachable from another state in at most two steps is a general enough inclusion criterion. Note that we do not require 2-step reachability from an initial state, but rather from *any* state, hence this set over-approximates the set of reachable states.

4.5 Implementation and Experimental Results

We implemented an automatic verifier for array programs based on our approach, and applied it successfully to verify natural properties of a few interesting array-manipulating programs.

Base case. We discharged the base case of the induction (the verification on the base states) using KLEE [27]—a state-of-the-art symbolic execution [25] engine. It took KLEE less than one tenth of a second to verify the correctness of each program in our benchmarks on the states in its base. This part of our verification approach is standard, and we discuss it no further; in the rest of this section we focus on the generation of the squeezing functions.

4.5.1 Implementation

The generate step and phase 1 of the test step of the squeezer synthesizer were implemented using a standalone C++ application that generates all $\curlyvee$ candidates with an AST of depth three. Each squeezer was tested on a pre-prepared state bank and every time a squeezer passed the tests it was immediately passed on to phase 2. The state bank contained states with arrays of length five or less. For each benchmark, we used up to 24,386 states with randomly selected array contents. The number of states was determined as follows: Suppose the program state is comprised of k variables and an array of size n. We randomly selected p elements that can populate the array: $p = \{'a','b',0\}$ for string manipulating procedures and $p = \{-4, -2, 9, 100, 200\}$ for programs that manipulate integer arrays. We determined the number of test states according to the following formula: $|p|^{n \cdot k}/df$, where df is an arbitrary dilution factor used to reduce the number of states from thousands to hundreds. (In our experiments, $df = 17$.)

The second and third phases were implemented using Z3 [39], a state of the art SMT solver. We chose to use the theory of sequences, since its API allows for a straightforward definition of the operation remove(arr,i) (see Figure 4.6). In practice, the sequence solver proved to be overall more effective than a corresponding encoding using the more mature array solver. In that aspect, it is worth mentioning that verifying fault preservation on its own *is* faster with the theory of arrays. We conjecture that this is because the specification has quantifiers while the other requirements can be verified using quantifier-free reasoning.

The transition relation was manually encoded in SMT-LIB2 format. However, it should be straightforward to automate this step.

4.5.2 Experimental Evaluation

We evaluated our technique by verifying a few array-manipulating programs against their expected specifications. The experiments were executed on a laptop with Intel i7-8565 CPU (4 cores) with 16GB of RAM running Ubuntu 18.04.

			Phase 1			Phase 2		Phase 3	Total Time	QUIC3
Program	B	# Cand	|Bank|	Test	Time	BMC	Time	Time	G&T+KLEE	Time
`strnchr`	2	80	356	29	0.004	1	0.12	0.16	0.28 + 0.07	0.32
`strncmp`	2	980	76	196	0.02	1	7.2	154.48	161.70 + 0.05	0.19
`max_ind`	2	8000	368	10	0.18	2	1.86	4.44	4.73 + 0.05	0.11
`min_ind`	2	8000	257	9	0.26	2	2.1	16.86	17.21 + 0.05	0.09
`sum_bidi`	2	6328125	4602	1200	2.18	1	0.57	0.61	3.36 + 0.05	t.o.
`is_sorted`	4	900	25736	764	4.37	1	0.59	0.67	5.63 + 0.06	0.15
`long_pref`	3	6480	24386	4696	22.93	1	1.25	0.89	25.07 + 0.05	t.o.

Table 4.1: Experimental results (end-to-end). Time in seconds. G&T is a shorthand for Generate&Test

Benchmarks. We ran our experiments on seven array-manipulating programs: `strnchr` looks for the first appearance of a given character in the first n characters of a string buffer. `strncmp` compares whether two strings are identical up to their first n characters or the first zero character. `max_ind` (resp. `min_ind`) looks for the index of the maximal (resp. minimal) element in an integer array. `sum_bidi` is our running example. `is_sorted` checks if the elements of an array are sorted in an increasing order. `long_pref` is looking for the longest prefix of an array comprised of either a monotonically increasing or a monotonically decreasing sequence.

The user supplies predicates that are used when synthesizing each squeezer. These were selected based on understanding what the program does and the operations it uses internally. *E.g.*, for `strncmp` equality comparisons between same-index elements of the two input arrays are used (`s1[0]==s2[0]` etc.), as well as comparison with constant 0; for `long_pref`, order comparisons (`s1[1]<=s1[2]` etc.) between different elements of the same array are used instead.

Results. Table 4.1 describes the end-to-end running times of our verifier, i.e., the time it took our tool to establish the correctness of each example. In this experiment, every candidate squeezer was tested before the next squeezer was generated. The table shows the time it took the synthesizer to find the first simulation-inducing squeezer plus the time it took to establish the correctness of the programs on the states in the base using KLEE (Total Time). The table also compares our verifier to QUIC3 [68], an automatic synthesizer of loop invariants. In general, when both tools where able to prove that the analyzed procedure is correct, QUIC3 was somewhat faster, and in the case of `strncmp` much faster. However, on two of our benchmarks QUIC3 timed out (1 hour) whereas our tool was able to prove them correct in less than 30 seconds.

	Phase 1				Phase 2			Phase 3		
Program	$\vert$Pos.$\vert$	Time	$\vert$Neg.$\vert$	Time	$\vert$Pos.$\vert$	Time	Time$_{\vert\text{Neg.}\vert}$	$\vert$Pos.$\vert$	Time	Time$_{\vert\text{Neg.}\vert}$
`strnchr`	1	ϵ	9	ϵ	1	0.94	–	1	0.98	–
`strncmp`	3	ϵ	36	ϵ	3	14.29	–	3	154.48	–
`max_ind`	11	ϵ	3	ϵ	2	0.78	1.08	1	31.00	0.41
`min_ind`	11	ϵ	7	ϵ	2	0.91	1.19	1	16.00	0.43
`sum_bidi`	12	ϵ	1	0.05	1	0.56	0.69	1	0.61	–
`is_sorted`	1	ϵ	18	ϵ	1	0.59	–	1	0.67	–
`long_pref`	2	ϵ	74	ϵ	1	1.03	1.22	1	0.89	–

Table 4.2: Experimental results. Time in seconds. $\epsilon \leq 0.0001$

Table 4.1 also provides more detailed statistics regarding the experiments: The rank of the base states (B), the total number of possible candidates based on the supplied predicates and the bound on the depth of the AST (# Cand), and a more detailed view of each phase in the testing step. For phase 1, it reports the number of states in the pre-prepared state bank (|Bank|), the number of squeezers tested until a simulation-inducing one was found (Test), and the total time spent to test these squeezers (Time). For phase 2, it reports the number of candidates which passed phase 1 and survived bounded verification (BMC) and the time spent in this phase (Time). For phase 3, we report how many simulation-inducing squeezers were found, and the time it took to apply full verification.

In all our experiments except of `max/min_ind` only the simulation-inducing squeezers passed bounded verification. In the latter case, a squeezer passed BMC due to the use of arrays of size at most five where the cells $a[2]$ and $a[n-2]$ are adjacent. Had we increased the array bound to six, these false positives would have been eliminated by the bounded verification.

Table 4.2 provides average times required to pass all the generated squeezers through the testing pipeline. For phase 1, it reports the number of squeezers which passed (Pos) resp. failed (Neg) testing against the randomly generated states and the average time it took to test the squeezers in each category (Time). The table reports the statistics pertaining to phase 2 and 3 in a similar manner, except that it omits the number of squeezers which failed the phase as this number can be read off the number of squeezers which reached this phase.

Table 4.3 shows some of the automatically generated squeezers. We obtained a single simulation-inducing squeezer in all of our tests except for `strncmp` where three

Program	**Squeezer**
`strchr(c)`	`if ( s[0] == c || s[0]==0 ) remove(s,1) else remove(s,0)`
`strncmp`	(1) `if (s1[0] == s2[0] && s1[0] != 0) remove(s1,0); remove(s2,0)`
	`else remove(s1,1); remove(s2,1)`
	(2) `if (s1[0] == s2[0] && s2[0] != 0) remove(s1,0); remove(s2,0)`
	`else remove(s1,1); remove(s2,1)`
	(3) `if (s1[0] != s2[0]) || (s1[0] == 0 && s2[0] == 0))`
	`remove(s1,1); remove(s2,1)`
	`else remove(s1,0); remove(s2,0)`
`max_ind`	`if (s[n-2] <= s[n-1]) remove(s,n-2) else remove(s,n-1)`
`is_sorted`	`if (s[n-3]<=s[n-2]<=s[n-1]) remove(s,n-1) else remove(s,n-4)`
`long_pref`	`if ((s[0]<=s[1]<= s[2]) || (s[0]>s[1]>s[2])) remove(s,0)`
	`else remove(s,n-1)`

Table 4.3: Syntesized squeezers. n is the size of the input array

squeezers were synthesized. The three differ only syntactically by the condition of the `if` statements. However, semantically, the three conditions are equivalent. Thus, improving the symmetry-detection optimizations to include equivalence up-to-de morgan rules would have filtered out two of the three squeezers.

4.6 Related Work

Automatic verification of infinite-state systems, *i.e.* , systems where the size of an individual state is unbounded such as numerical programs (where data is considered unbounded), array manipulating programs (where both the length of the array and the data it contains may be unbounded), programs with dynamic memory allocation (with unbounded number of dynamically-allocatable memory objects), and parameterized systems (where, in most cases, there is an unbounded number of instances of finite subsystems) is a long standing challenge in the realm of formal methods.

Well structured transition systems. Well structured transition systems (WSTS) [2, 3, 47] are a class of infinite-state transition systems for which safety verification is decidable, with a backward reachability analysis being a decision procedure. In these transitions systems, the set of states is accompanied by a well-quasi order that induces a simulation relation: a state is simulated by those that are “larger” than it. As a result, the set of backward-reachable states is upward closed. The simulation-inducing well-quasi order used in WSTS resembles our condition of a simulation-inducing squeezer. However, there are several fundamental differences:

- The order underlying our technique is required to be well-founded, which is a strictly weaker requirement than that of a well-quasi order;
- The simulation-inducing requirement requires each state to be simulated by its squeezed version, which has a *lower* rank rather than greater; further, a state need not be simulated by *every* state with a lower rank; accordingly, the set of backward-reachable states need not be upward (nor downward) closed.
- Our procedure is not based on backward (or any other form of) reachability analysis.

Reductions. Cutoff-based techniques, e.g., [42], reduce model checking of unbounded parameterized systems to model checking for systems of size (up to) a small predetermined cutoff size. Verification based on dynamic cut-offs [4, 83] also considers parameterized systems but employs a verification procedure which can dynamically detect cut-off points beyond which the search of the state space need not continue. Invisible invariants [114, 155] are used to verify unbounded parameterized systems in a bounded way. The idea is to use the standard deductive invariance rule for proving invariance properties but consider only bounded systems for discharging the verification conditions, while ensuring that they hold for the unbounded system. The approach provides (i) a heuristic to generate a candidate inductive invariant for the proof rule, and (ii) a method to validate the premises of the proof rule once a candidate is generated [155].

Similar reductions were applied to array programs–a particular form of parameterized systems but with unbounded data–as we consider in this work. For example, in [92], *shrinkable* loops are identified as loops that traverse large or unbounded arrays but may be soundly replaced by a bounded number of nondeterministically chosen iterations; and in [104], abstraction is used to replace reasoning about unbounded arrays and quantified properties by reasoning about a bounded number of array cells.

A fundamental difference between our approach and these works is that we do not reduce the problem to a bounded verification problem. Instead, we generate verification conditions which amount to a proof by induction on the size of the system. In fact, from the perspective of deductive verification, our work can be seen as introducing a new induction scheme.

Loop invariant inference. Arguably, inference of loop invariants is the ubiquitous approach for automatic verification of infinite-size systems. Recent research efforts in the area have concentrated around inference of quantified invariants, in particular, the search for universal loop invariants is a central issue.

Classical predicate abstraction [15, 59] has been adapted to quantified invariants by extending predicates with *skolem* (fresh) variables [48, 94]. This is sufficient for discovering complex loop invariants of array manipulating programs similar to the simpler programs used in our experiments.

A research avenue that has received ongoing popularity is the use of constrained Horn clauses (CHCs) to model properties of transition systems which have been used for inference of universally quantified invariants [20, 67, 103] by limiting the quantifier nesting in the loop invariant being sought. In [44], universally quantified solutions (inductive invariants) to CHCs are inferred via syntax-guided synthesis.

Another active research area is Model-Checking Modulo Theories (MCMT) [53] which extends model checking to array manipulating programs and has been used for verifying heap manipulating programs and parameterized systems (e.g., [34]) using quantifier elimination techniques. For example, in SAFARI [9] (and later BOOSTER [10]), the theory of arrays [24] is used to construct a QF proof of bounded safety which is generalized by universally quantifying out some terms.

IC3 [21] extends predicate abstraction into a framework in which the predicate discovery is directed by the verification goal and heuristics are used to generalize proofs of bounded depth execution to inductive invariants. UPDR [86] and QUIC3 [68] extend IC3 to quantified invariants. UPDR focuses on programs specified using the Effectively PRopositional (EPR) fragment of *uninterpreted* first order logic (e.g., without arithmetic) for which quantified satisfiability is decidable. As such, UPDR does not deal with quantifier instantiation. QUIC3 uses model based projection and generalizations based on bounded exploration.

Like these techniques we also use heuristics to overcome the unavoidable undecidability barrier. In our case, this amounts to the selection of the squeezing function. In contrast to all the aforementioned approaches, our technique does not rely on the inference of loop invariant but rather proves programs correct by induction on the size (rank) of their states.

We note that we do not position our technique as a replacement to automatic inference of loop invariants but rather as a complementary approach. Indeed, while some tricky properties can be easily verified by our approach, e.g., the postcondition of `sum_bidi`, a property which we believe no other automatic technique can deduce, other properties which are simple to establish using loop invariants, e.g., that variable `i` is always in the range $0..n-1$, are surprisingly challenging for our technique to establish.

Recurrences. Other approaches represent the behavior of loops in array-programs via recurrences defined over an explicit loop counter, and use these recurrences to directly verify post-conditions with universal quantification over the array indices. In [116] this is done by customized instantiation schemes and explicit induction when necessary. In [28], verification is done by identifying a relation between loop iterations (characterized by the loop counter) and the array indices that are affected by them, and verifying that the post-condition holds for these indices. Similarly to our approach, these works do not rely on loop invariants, but they do not allow to verify global properties over the arrays, such as the postcondition of `sum_bidi`.

Program synthesis. The inference we use for $\curlyvee$ is indeed a form of program synthesis, as was alluded to in Section 4.2 by representing $\curlyvee$ via pseudo-code. In particular, *syntax-guided synthesis* (SyGuS) [13] is the domain of program synthesis where the target program is derived from a programming language according to its syntax rules. [43, 81, 144, 146] all fall within this scope.

Sketching is a common feature of SyGuS. The term is inspired by Sketch [134], referring to the practice of giving synthesizers a program skeleton with a missing piece or pieces. This uses domain knowledge to reduce the size of the candidate space. It is quite common to use a domain-specific language (DSL) for this purpose [76, 133, 137, 139, 147]. [111] restricts programs by typing rules in addition to just syntax. [64] develops it further by restricting how operators may be composed. Our synthesis procedure (Section 4.4) follows the same guidelines: the domain of array-scanning programs dictates the constructed space of squeezer functions, and moreover, inspecting the analyzed program allows for more pruning by (i) matching index variables to array variables and (ii) focusing on operators and literal values occurring in the program. This early pruning is responsible for the feasibility of our synthesis procedure, which apart from that is rather naive and does not facilitate clever optimizations such as equivalence reduction [45, 111].

4.7 Conclusions

At the current state of affairs in automatic software verification of infinite state systems, the scene is dominated by various approaches with a common aim: computing over-approximations of unbounded executions by means of inferring loop invariants. Indeed, *abstract interpretation* [35], *property-directed reachability* [21], unbounded model checking [99], or template-based verification [138] can be seen as different techniques for computing such approximations by finding inductive loop invariants which are tight enough not to intersect with the set of bad behaviors. Experience has shown that these invariants are frequently quite hard to come by, even for seemingly simple and innocuous program, both automatically and manually. The purpose of this chapter is to suggest an alternative kind of correctness witness, which may be more amenable to automated search. We successfully applied our novel verification technique to array programs and managed to prove programs and properties which are beyond the ability of existing automatic verifiers. We believe that our approach can be combined with standard techniques to give rise to a new kind of hybrid techniques, where, e.g., a *partial* loop invariant is used as a baseline — verified via standard techniques — and is then *strengthened* to the desired safety property via squeezer-based verification.

Chapter 5

Size reduction for complexity analysis

This chapter is based on the results published in [80].

5.1 Introduction

Cost analysis is the problem of estimating the resource usage of a given program, over all of its possible executions. It complements functional verification—of safety and liveness properties—and is an important task in formal software certification. When used in combination with functional verification, cost analysis ensures that a program is not only correct, but completes its processing in a reasonable amount of time, uses a reasonable amount of memory, communication bandwidth, etc. In this work we focus on run-time complexity analysis. While the area has been studied extensively, e.g., [7, 11, 22, 37, 40, 49, 65, 73, 149], the general problem of constraining the number of iterations in programs containing loops with arbitrary termination conditions remains hard.

A prominent approach to computing upper bounds on the time complexity of a program identifies a well-founded numerical measure over program states that decreases in every step of the program, also called a *ranking function*. In this case, an upper bound on the measure of the initial states comprises an upper bound on the program's time complexity. Finding such measures manually is often extremely difficult. The *cost relations* approach, dating back to [149], attempts to automate this process by using the control flow graph of the program to extract recurrence formulas that characterize

```
void binary_counter(unsigned int n) {
  unsigned int c[n];
  memset(c,0,n*sizeof(unsigned int));
  int i=0;
  while (i<n) {
    if (c[i]==1) /*scan 1-prefix*/{c[i]=0;i++;          }
    else         /*increment    */{c[i]=1;i=0;print(c);}
  }}
```

Figure 5.1: A program that produces all combinations of n bits.

this measure. Roughly speaking, the recurrences relate the measures (costs) of adjacent nodes in the graph, taking into account the cost of the step between them. In this way, the cost relations track the evolution of the measure between *every* pair of consecutive states along the executions of the program.

One limitation of cost relations is the need to capture the number of steps remaining for execution in *every* state, that is, all intermediate states along all executions. If the structure of the state is complex, this may require higher order expressions, e.g., summing over an unbounded number of elements. As an example, consider the program in Figure 5.1 that implements a binary counter represented by an array of bits.

In this case, a ranking function that decreases between every two consecutive iterations of the loop, or even between two iterations that print the value of the counter, depends on the *entire* content of the array. Attempting to express a ranking function over the scalar variables of this program is analogous to abstracting the loop as a finite-state system that ignores the content of the array, and as such contains transition cycles (e.g. the abstract state $\langle n \mapsto n_0, i \mapsto 0\rangle$, obtained by projecting the state to the scalar variables only, repeats multiple times in any trace)—meaning that no strictly decreasing function can be defined in this way. Similarly, any attempt to consider a bounded number of bits will encounter the same difficulty.

In this paper, we propose a novel approach for extracting recurrence relations capturing the time complexity of an imperative program, modeled as a transition system, by relating whole traces instead of individual states. The key idea is to relate a trace to (one or more) shorter traces. This allows to formulate a recurrence that resolves to the length of the trace and recurs over the values at the initial states only. We sidestep the need to take into account the more complex parts of the state that change along the trace (e.g., in the case of the binary counter, the array is initialized with zeros).

Our approach relies on the notion of *state squeezers* [78], previously used exclusively for the verification of safety properties. We present a novel aspect where the same squeezers can be used to determine complexity bounds, by replacing the safety property check with trace length judgements.

Squeezers provide a means to perform induction on the "size" of (initial) states to prove that all reachable states adhere to a given specification. This is accomplished by attaching *ranks* from a well-founded set to states, and defining a *squeezer function* that maps states to states of a lower rank. Note that the notion of a rank used in our work is distinct from that of a ranking function, and the two should not be confused; in particular, a rank is not required to decrease on execution steps. In chapter 4, squeezers were utilized for safety verification: the ability to establish safety is achieved by having the squeezer map states in a way that forms a (relaxed form of) a *simulation relation*, ensuring that the traces of the lower-rank states simulate the traces of the higher rank states. Due to the simulation property, which is verified locally, safety over states with a *base* rank, carries over (by induction over the rank) to states of any higher rank.

In this chapter, we use the construction of well-founded ranks and squeezers to define a *recurrence formula* representing (an upper bound on) the time complexity of the procedure being analyzed. We do so by expressing the complexity (length) of traces in terms of the complexity of lower-rank traces. This new setting raises additional challenges: it is no longer sufficient to relate traces to lower-rank traces; we also need to *quantify the discrepancy* between the lengths of the traces, as well as between their ranks. This is achieved by a certain form of simulation that is parameterized by *stuttering shapes* (for the lengths) and by means of a *rank bounding function* (for the ranks). Furthermore, while [78] limits each trace to relate to a *single* lower-rank trace, we have found that it is sometimes beneficial to employ a *decomposition* of the original trace into *several* consecutive *trace segments*, so that each segment corresponds to *some* (possibly different) lower-rank trace.The segmentation simplifies the analysis of the length of the entire trace, since it creates sub-analyses that are easier to carry out, and the sum of which gives the desired recurrence formula. This also enables a richer set of recurrences to be constructed automatically, namely non-single recurrences (meaning that the recursive reference may appear more than once on the right hand side of the equation).

The base case of the recurrence is obtained by computing an upper bound on the time complexity of base-rank states. This is typically a simpler problem that may be addressed, e.g., by symbolic execution due to the bounded nature of the base. The solution to the recurrence formula with the respective base case soundly overapproximates the time complexity of the procedure.

We show that, conceptually, the classical approach for generating recurrences based on ranking functions can be viewed as a special case of our approach where the squeezer maps a state to its immediate successor. The real power of our approach is in the freedom to define other squeezers, producing simpler recursions, and avoiding the need for complex ranking functions.

Our use of squeezers for extracting recurrences that bound the complexity of imperative programs is related to the way analyses for functional programs (e.g. [74]) use the term(s) in recursive function calls to extract recurrences. The functional programming style coincidentally provides such candidate terms. The novelty of our approach is in introducing the concept of a squeezer explicitly, leading to a more flexible analysis as it does not restrict the squeezer to follow specific terms in the program. In particular, this allows reasoning over space in imperative programs as well.

The main results of this work can be summarized as follows:

- We propose a novel technique for run-time complexity analysis of imperative programs based on state squeezers. Squeezers, together with rank-bounding functions, are used for extracting recurrence relations whose solutions overapproximate the length of executions of the input program.

- We formalize the notions of *state squeezers*, *partitioned simulation* and *rank bounding functions* that underlie the approach, and establish conditions that ensure soundness of the recurrence relations.

- We demonstrate that squeezers and rank bounding functions can be efficiently synthesized and verified, due to their compactness, especially relative to explicit ranking functions.

- We implemented our approach and applied it successfully to several small but intricate programs, some of which could not have been handled by existing techniques.

5.2 Overview

In this section we give a high level description of our technique for complexity analysis using the binary counter example in Figure 5.1.

Example: Binary counter The procedure in Figure 5.1 receives as an input a number n of bits and iterates over all their possible values in the range $0...2^n - 1$. The "current" value is maintained in an array c which is initialized to zero and whose length is n. $c[0]$ represents the least significant bit. The loop scans the array from the least significant bit forward looking for the leftmost 0 and zeroing the prefix of 1s. As soon as it encounters a 0, it sets it to 1 and starts the scan from the beginning. The program terminates when it reaches the end of the array ($i = n$), all array entries are zeros, and the last value was $111\ldots$; at this point all the values have been enumerated.

Existing analyses All recent methods that we are aware of (such as [6, 49, 74]) fail to analyze the complexity of this procedure (in fact, most methods will fail to realize that the loop terminates at all). One reason for that is the need to model the contents of the array whose size in unknown at compile time. However, even if data *were* modeled somehow and taken into account, finding a ranking function, which underlies existing approaches, is hard since this function is required to decrease between *any* two consecutive iterations along *any* execution. Here for instance, to the best of our knowledge, such a function would depend on an unbounded number of elements of the array; it would need to extract the current value as an integer, along the lines of $\sum_{j=0}^{n-1} c[j] \cdot 2^j$. The use of a ranking function for complexity analysis is somewhat analogous to the use of inductive invariants in safety verification. Both are based on induction over time along an execution. This work is inspired by the technique presented in chapter 4, showing that verification can also be done when the induction is performed on the size (*rank*) of the state rather than on the number of iterations, where the size of the state may correspond, e.g., to the size of an unbounded data structure. We argue that similar concepts can be applied in a framework for complexity classification. That is, we try to infer a recurrence relation that is *based on the rank* of the state and correlates the lengths of *complete* executions—executions that start from an initial state—of different ranks. This sidesteps the need to express the length of *partial* executions, which start from intermediate states. While the approach applies to bounded-state systems as well, its benefits become most apparent when the program contains a-priori unbounded stores, such as arrays.

Our approach. Roughly speaking, our approach for computing recurrence formulas that provide an upper bound on the complexity of a procedure is based on the following ingredients:

- A *rank* function $r : \mathit{init} \to X$ that maps initial states to ranks from a well founded set $(X, \prec)$ with base B. Intuitively, the rank of the initial state governs the time complexity of the entire trace, and we also consider it to be the rank of the trace. As we shall soon see, this rank can be significantly simpler than a ranking function.
- A *squeezer* $\curlyvee : \Sigma \to \Sigma$ that maintains (some variant of) a simulation relation, thus ensuring a bona fide correspondence between higher-rank traces and lower-rank traces through correspondence between states.
- A *trace partition* $p_d : \Sigma \to [1..d]$ that maps each state to a segment-identifier $i \in [1..d]$, and induces a decomposition of a trace into *segments*, allowing $\curlyvee$ to map each of them to a separate, lower-rank *mini-trace*.
- A *rank-bounding* function $\hat{\curlyvee} : X \times [1..d] \to X$ that provides an upper bound on the rank of the initial states of the d mini-traces based on the rank of the higher-rank trace. (The rank is *not* required to be uniform across mini-traces).

All of these ingredients are synthesized automatically, as we discuss in Section 5.4. Next, we elaborate on each of these ingredients, and illustrate them using the binary counter example. We further demonstrate how we use these ingredients to find recurrence formulas describing (an upper bound on) the complexity of the program.

Some notations We adopt a standard encoding of a program as a transition system over a state space Σ, with a set of initial states $\mathit{init} \subseteq \Sigma$ and transition function $\mathit{tr} : \Sigma \to \Sigma$, where a transition corresponds to a loop iteration. We use $\mathit{reach} \subseteq \Sigma$ to denote the set of reachable states, $\mathit{reach} = \{\sigma \mid \exists \sigma_0, k.\ \mathit{tr}^k(\sigma_0) = \sigma \wedge \sigma_0 \in \mathit{init}\}$.

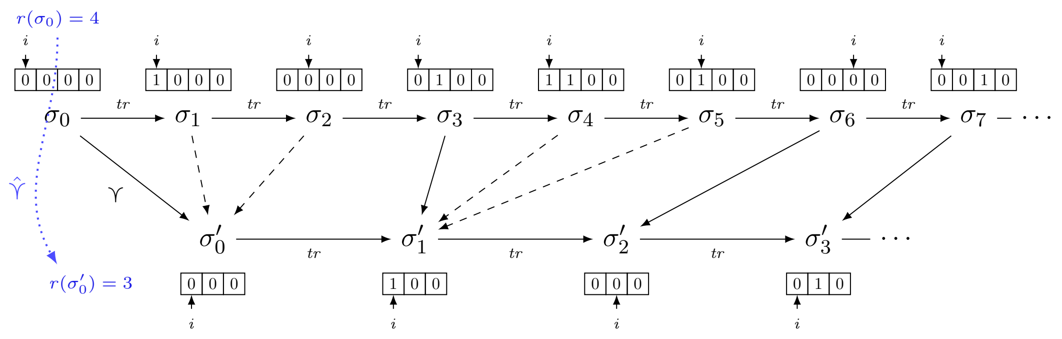
Defining the rank of a state Ranks are taken from a well founded set $(X, \prec)$ with a basis $B \subseteq X$ that contains all the minimal elements of X. The rank function,

$r : \mathit{init} \to X$, aims to abstract away irrelevant data from the (initial) state that does *not* effect the execution time, and only uses state "features" that do. When proper ranks are used, the rank of an initial state is all that is needed to provide a tight bound on its trace length. Since ranks are taken from a well founded set, they can be recursed over. In the binary counter example, the chosen rank is n, namely, the rank function maps each state to the size of the array. (Notice that the rank does not depend on the contents of the array; in contrast, bounding the trace length from any intermediate state, and not just initial states, would have required considering the content of the array). Given the rank function, our analysis extracts a recurrence formula for the complexity function $\mathit{comp}_x : X \to \mathbb{N} \cup \{\infty\}$ that provides an upper bound on the number of iterations of tr based on the rank of the *initial states*. In our exposition, we sometimes also refer to a time complexity function over states, $\mathit{comp}_s : \mathit{init} \to \mathbb{N} \cup \{\infty\}$, which is defined directly on the (initial) states, as the number of iterations in an execution that starts with some $\sigma_0 \in \mathit{init}$.

Defining a squeezer The squeezer $\curlyvee : \Sigma \to \Sigma$ is a function that maps states to states of lower-rank traces (where the rank of a trace is determined by the rank of its initial state), down to the base ranks B. Its importance is in defining a correspondence between higher-rank traces and lower-rank ones that can be verified locally, by examining individual states rather than full traces. The kind of correspondence that the squeezer is required to ensure affects the flexibility of the approach and the kind of recurrence formulas that it may yield. To start off, consider a rather naive squeezer that satisfies the following local properties:

- rank decrease of non-base initial states: $\sigma_0 \in \mathit{init} \wedge r(\sigma_0) \notin B \Rightarrow r(\curlyvee(\sigma_0)) \prec r(\sigma_0)$, and
- initial anchor: $\sigma_0 \in \mathit{init} \Rightarrow \curlyvee(\sigma_0) \in \mathit{init}$,
- k-step: $\sigma \in \mathit{reach} \Rightarrow \exists k.\ \mathit{tr}(\curlyvee(\sigma)) = \curlyvee(\mathit{tr}^k(\sigma))$.

As an example, the squeezer we consider for the binary counter program is rather intuitive: it removes the least significant bit ($c[0]$), and adjusts the index i accordingly. Doing so yields a state with rank $r(\curlyvee(\sigma_0)) = r(\sigma_0) - 1$. Figure 5.2 shows the correspondence between a 4-bit binary counter, and a 3-bit one. The figure illustrates the



$\Upsilon(\langle n, i, c\rangle) = \langle n \dot{-} 1, i \dot{-} 1, c[1{:}]\rangle \qquad \hat{\Upsilon}(n) = n \dot{-} 1 \qquad (x \dot{-} y = \max\{0, x - y\})$

Figure 5.2: Correspondence between two traces of the binary counter program. Squeezer removes the leftmost array entry, that represents the least significant bit. The rank is the array size, i.e., four on the upper trace and three on the lower one. The simulation includes only 1-,2- and 3-steps, so the length of the upper trace is at most three times that of the lower trace, yielding an overall complexity bound of $O(3^n)$.

simulation k-step property for $k = 1, 2, 3$: σ_0 and σ_3 are $(3, 1)$-stuttering, σ_1 and σ_4 are $(2, 1)$-stuttering, and σ_2, σ_5 and σ_6 are $(1, 1)$-stuttering.

The simulation property induces a correlation between a higher rank trace τ and a lower rank one τ', such that every step of τ' is matched by k steps in τ. Whenever a state σ satisfies the k-step property, we will refer to it as being $(k, 1)$-*stuttering*. (We usually only care about the smallest k that satisfies the property for a given σ.) Now suppose that there exists some $\widehat{k} \in \mathbb{N}^+$ such that for every trace $\tau(\sigma_0)$ and every state $\sigma \in \tau(\sigma_0)$, σ is $(k, 1)$-stuttering with $1 \le k \le \widehat{k}$. This would yield the following complexity bound:

$$comp_s(\sigma_0) \le \widehat{k} \cdot comp_s(\Upsilon(\sigma_0)). \tag{5.1}$$

Base case What should happen if we repeatedly apply Υ to some initial state σ_0, each time obtaining a new, lower-rank trace? Since $r(\Upsilon(\sigma_0)) \prec r(\sigma_0)$, and since $(X, \prec)$ is well-founded, we will eventually hit some state of *base rank*:

$$\Upsilon(\Upsilon(\ldots(\sigma_0))\ldots) = \sigma_0^\circ \quad \text{such that} \quad r(\sigma_0^\circ) \in B$$

Hence, if we know the complexity of the initial states with a base rank, we can apply Equation (5.1) iteratively to compute an upper bound of the complexity of *any* initial state.

How many steps will be needed to get from an arbitrary initial state σ_0 to σ_0°? Clearly, this depends on the rank, and the way in which Υ decreases it.

Consider the binary counter program again, with the rank $r(\sigma) = n$. $(\mathbb{N}, <)$ is well-founded, with a single minimum 0. If we define, e.g., $B = \{0, 1\}$, we know that the length of any trace with $n \in B$ is bounded by a constant, 2. (Bounding the length of traces starting from an initial state σ_0 where $r(\sigma_0) \in B$ can be done with known methods, e.g., symbolic execution). Since the rank decreases by 1 on each "squeeze", we get the following exponential bound:

$$comp_s(\sigma_0) \leq 2 \cdot 3^{n-1} = O(3^n) \tag{5.2}$$

The last logical step, going from (5.1) to (5.2), is, in fact, highly involved: since Equation (5.1) is a mapping of *states*, solving such a recurrence for arbitrary Υ cannot be carried out using known automated methods. Instead, we implicitly used the rank of the state, n, to extract a recurrence over scalar values and obtain a closed-form expression. Let us make this reasoning explicit by first expressing Equation (5.1) in terms of $comp_x$ instead of $comp_s$:

$$comp_x(n) \leq \widehat{k} \cdot comp_x(n-1)$$

Here, $n-1$ denotes the rank obtained when squeezing an initial state of rank n. Unlike Equation (5.1), this is a recurrence formula over $(\mathbb{N}, <)$ that may be solved algorithmically, leading to the solution $comp_x(n) = O(3^n)$.

Surplus analysis Assuming the worst k for all the states in the trace can be too conservative; in particular, if there are only a few states that satisfy the $\widehat{k}$-step property, and all the others satisfy the 1-step property. In the latter case, if we know that at most b states in any one trace have $k > 1$, we can formulate the tighter bound:

$$comp_s(\sigma_0) \leq comp_s(\Upsilon(\sigma_0)) \ + \ \widehat{k} \cdot b \tag{5.3}$$

Incidentally, in the current setting of the binary counter program, the number of $\widehat{k}$-steps (3-steps) is *not* bounded. So we cannot apply the inequality (5.3) repeatedly on any trace, as the number of 3-steps depends on the initial state. However, we can improve the analysis by partitioning the trace to two parts, as we explain next.

Segments and mini-traces Note that both (5.1) and (5.3) "suffer" from an inherent restriction that the right hand side contains *exactly* one recursive reference. As such, they are limited in expressing certain kinds of complexity classes.

In order to get more diverse recurrences, including non-single recurrences, we propose an extension of the simulation property that allows more than one lower-rank trace:

- *partitioned* simulation
- initial anchor: $\sigma_0 \in init \Rightarrow \Upsilon(\sigma_0) \in init$ *(same as before)*,
- k-step: $\sigma \in reach \Rightarrow \exists k.\ tr\big(\Upsilon(\sigma)\big) = \Upsilon\big(tr^k(\sigma)\big)$ *(same as before)* <u>*or*</u>

  $\Upsilon\big(tr(\sigma)\big) \in init$ *(switch)*

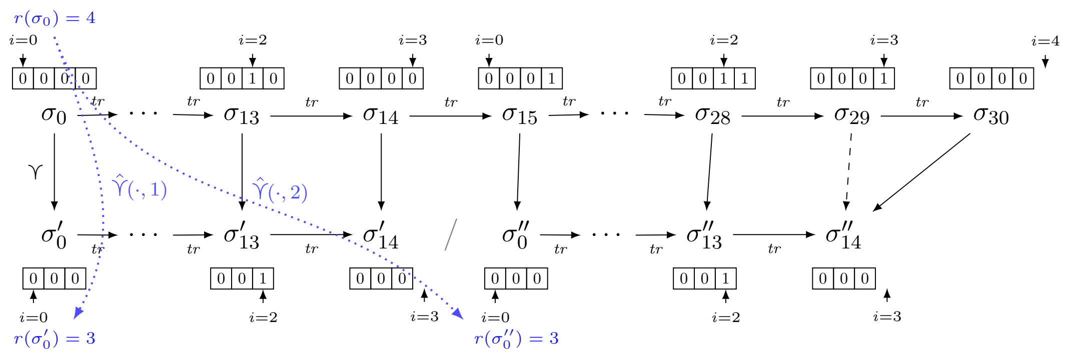
This definition allows a new mini-trace to start at any point along a higher-rank trace τ, thus marking the beginning of a new segment of τ. When this occurs, we call $tr(\sigma)$ a *switch state*. For the sake of uniformity, we also refer to all initial states $\sigma_0 \in init$ as switch states. Hence, each segment of τ starts with a switch state, and the mini-traces are the lower-level traces that correspond to the segments (these are the traces that start from $\Upsilon(\sigma_s)$, where σ_s is a switch state). The length of τ can now be expressed as the *sum* of lower-level mini-traces.

However, there are two problems remaining. First, we need to extend the "rank decrease of non-base initial states" requirement to any switch state in order to ensure that the ranks of all mini-traces are indeed lower. Namely, we need to require that if σ_s is any switch state in a trace from σ_0, then $r\big(\Upsilon(\sigma_s)\big) \prec r(\sigma_0)$.

Second, even if we extend the rank decrease requirement, this definition does not suggest a way to bound the number of correlated mini-traces and their respective ranks, and therefore suggests no effective way to produce an equation for $comp_s$ as before.

To sidestep the problem of a potentially unbounded number of mini-traces, we augment the definition of simulation with a *trace partition* function; to address the challenge of the rank decrease we use a *rank-bounding* function, which is responsible both for ensuring that the rank of the mini-traces decreases and for bounding their ranks.

Defining a partition We define a function $p_d : \Sigma \to \{1, \ldots, d\}$, parameterized by a constant d, called a *partition function*, that is weakly monotone along any trace



$\Upsilon\big(\langle n, i, c\rangle\big) = \langle n \mathbin{\dot{-}} 1, (i < n)\ ?\ i : i{-}1, c[{:}n{-}1]\rangle \qquad \hat{\Upsilon}(n) = n \mathbin{\dot{-}} 1$

Figure 5.3: An execution trace of the binary counter program that corresponds to two mini-traces of lower rank.

$(p_d(\sigma) \leq p_d(tr(\sigma)))$. This function induces a partition of any trace τ into (at most) d segments by grouping states based on the value of $p_d(\sigma)$. To ensure the segments and mini-traces are aligned, we require that switch states only occur at segment boundaries.

- *d-partitioned* simulation:
- initial anchor: $\sigma_0 \in \mathit{init} \Rightarrow \Upsilon(\sigma_0) \in \mathit{init}$ *(same as before)*,
- k-step: $\sigma \in \mathit{reach} \Rightarrow \exists k.\ tr\big(\Upsilon(\sigma)\big) = \Upsilon(tr^k(\sigma))$ *(same as before)* <u>or</u>

$$\Upsilon\big(tr(\sigma)\big) \in \mathit{init}\ \wedge\ p_d(\sigma) < p_d\big(tr(\sigma)\big) \qquad \textit{(segment switch)}$$

In our running example, let us change Υ so that it shrinks the state by removing the *most* significant bit instead of the least. This leads to a partition of the execution trace for $r(\sigma_0) = n$ into two segments, as shown in Figure 5.3. The partition function is $p_d = (i \geq n \mid\mid c[n-1])\ ?\ 2\ :\ 1$ (essentially, $c[n-1]+1$, except that the final state is slightly different). As can be seen from the figure, each segment simulates a mini-trace of rank $n-1$, with $k=1$ for all the steps except for the last step (at σ_{28}) where $k=2$. In this case, it would be folly to use the recurrence (5.1) with $\widehat{k} = 2$, since all the steps are 1:1 except one. Instead, we can formulate a tighter bound:

$$comp_s(\sigma_0) \leq comp_s(\sigma'_0) + comp_s(\sigma''_0) + 2$$

Where: $comp_s(\sigma'_0)$, $comp_s(\sigma''_0)$ are the lengths of the mini-traces, and 2 is the surplus from the switch transition $\sigma_{14} \to \sigma_{15}$ plus the 2-step at σ_{28}. In the case of this program, we know that $r(\sigma'_0) = r(\sigma''_0) = r(\sigma_0) - 1$, for any initial state σ_0, therefore, turning to

$comp_x$, we can derive and solve the recurrence $comp_x(n) = 2 \cdot comp_x(n-1) + 2$, which together with the base yields the bound:

$$comp_x(n) = 2^{n+1} - 2$$

Clearly, a general condition is required in order to identify the ranks of the corresponding initial states of the (lower-rank) mini-traces (and at the same time, ensure that they decrease).

Bounding the ranks of squeezed switch states This is not a trivial task, since as previously noted, the squeezed ranks could be different, and may depend on properties present in the corresponding switch states. To achieve this goal, once a partition function p_d is defined, we also define a rank-bounding function $\hat{\Upsilon} : X \times \{1, \ldots, d\} \rightarrow X$, where for any $\sigma_0 \in init$ and switch state σ_s, $\hat{\Upsilon}$ provides a bound for the rank of $\Upsilon(\sigma_s)$ based on that of σ_0:

$$r(\Upsilon(\sigma_s)) \preceq \hat{\Upsilon}\big(r(\sigma_0), p_d(\sigma_s)\big) \prec r(\sigma_0) \tag{5.4}$$

The rightmost inequality ensures that a mini-trace that starts from $\Upsilon(\sigma_s)$ is of lower-rank than σ_0, and as such extends the "rank decrease" requirement to all mini-traces. Based on this restriction, we can formulate a recurrence for $comp_x$ based on the initial rank $rank = r(\sigma_0)$, as follows:

$$comp_x(rank) \leq \sum_{i=1}^{d} comp_x\big(\hat{\Upsilon}(rank, i)\big) \; + \; (d-1) \; + \; \widehat{k} \cdot b \tag{5.5}$$

Where b, as before, is the number of k-steps for which $k > 1$, and $\widehat{k}$ is the bound on k ($k \leq \widehat{k}$). The expression $(d-1)$ represents the transitions between segments, and $\widehat{k} \cdot b$ represents the surplus of the $rank$-rank trace over the total lengths of the mini-traces.

It should be clear from the definition above, that $\hat{\Upsilon}$ is quite intricate. How would we compute it effectively? The rank decrease of the initial states and the simulation properties were *local* by nature, and thus amenable to validation with an SMT solver. The $\hat{\Upsilon}$ function is inherently *global*, defined w.r.t. an entire trace. This makes the property (5.4) challenging for verification methods based on SMT. To render this check more feasible with first-order reasoning, we introduce two special cases where the problem of checking (5.4) becomes easier: rank preservation and a single segment, explained next.

Taming $\hat{\curlyvee}$ with rank preservation To obtain rank preservation, we extend the rank function to all states (instead of just the initial states), and require that the rank is preserved along transitions. This is appropriate in some of the scenarios we encountered. For example, the binary counter illustration satisfies the property that along any execution $\{\sigma_i\}_{i=0}^{\infty}$, the rank is preserved: $r(\sigma_i) = r(\sigma_{i+1})$. Rank preservation means that given a switch state σ_s of an arbitrary segment i, we know that $r(\sigma_s) = r(\sigma_0)$. Once this is set, $\hat{\curlyvee}$ only needs to overapproximate the rank of $\curlyvee(\sigma)$ in terms of the rank of the same state σ.

Taming $\hat{\curlyvee}$ with a single segment In this case, checking (5.4) reduces to a single check of the initial state, which is the only switch state. It turns out that the restriction to a single segment is still expressive enough to handle many loop types.

Putting it all together Theoretically, r, $\curlyvee$, p_d, and $\hat{\curlyvee}$ can be manually written by the user. However, this is a rather tedious task, that is straightforward enough to be automated. We observed that all the aforementioned functions are simple enough entities, that can be expressed through a strict syntax using first order logic. Similar to the analysis in chapter 4, we apply a generate-and-test synthesis procedure to enumerate a space of possible expressions representing them. This process is explained in Section 5.4.

5.3 Complexity Analysis based on Squeezers

In this section we develop the formal foundations of our approach for extracting recurrence relations describing the time complexity of an imperative program based on state squeezers. We present the ingredients that underly the approach, the conditions they are required to satisfy, and the recurrence relations they induce. In the next section, we explain how to extract the recurrences automatically. Given the recurrence relation, a dedicated (external) tool may be applied to end up with a closed formula, similar to [7].

We use *transition systems* to capture the semantics of a program.

Definition 9 (Transition Systems). *A transition system is a tuple $(\Sigma, init, tr)$, where Σ is a set of* states, *$init \subseteq \Sigma$ is a set of* initial states *and $tr : \Sigma \to \Sigma$ is a* transition function *(rather than a transition relation, since only deterministic procedures are considered). The set of* terminal states $F \subseteq \Sigma$ *is implicitly defined by* $tr(\sigma) = \sigma$*. An* execution trace *(or a* trace *in short) is a finite or infinite sequence of states* $\tau = \sigma_0, \sigma_1, \ldots$ *such that* $\sigma_{i+1} = tr(\sigma_i)$ *for every* $0 \leq i < |\tau|$*. A state* $\sigma \in \Sigma$ *defines an execution trace* $\tau(\sigma) = \{tr^i(\sigma)\}_{i \in \mathbb{N}}$*. Whenever there exists an index* $0 \leq k \leq |\tau|$ *s.t.* $\sigma_k \in F$*, we truncate* $\tau(\sigma)$ *into a finite trace* $\{tr^i(\sigma)\}_{i=0}^{k}$*, where* k *is the minimal such index. The trace is* initial *if it starts from an initial state, i.e.,* $\sigma \in init$*. Unless explicitly stated otherwise, all traces we consider are initial. The set of* reachable states *is* $reach = \{\sigma \in \Sigma \mid \exists \sigma_0 \in init \,.\, \sigma \in \tau(\sigma_0)\}$*.*

Roughly, to represent a program by a transition system, we translate it into a single loop program, where *init* consists of the states encountered when entering the loop, and transitions correspond to iterations of the loop.

In the sequel, we fix a transition system $(\Sigma, init, tr)$ with a set F of terminal states and a set *reach* of reachable states.

Definition 10 (Complexity over states)**.** *For a state* $\sigma \in \Sigma$*, we denote by* $comp_s(\sigma)$ *the number of transitions from* σ *to a terminal state along* $\tau(\sigma)$ *(the trace that starts from* σ*). Formally, if* $\tau(\sigma)$ *does not include a terminal state, i.e., the procedure does* not *terminate from* σ*, then* $comp_s(\sigma) = \infty$*. Otherwise:*

$$comp_s(\sigma) = \min\{k \in \mathbb{N} \mid tr^k(\sigma) \in F\}.$$

The complexity function of the program maps each initial state $\sigma_0 \in init$ *to its time complexity* $comp_s(\sigma_0) \in \mathbb{N} \cup \{\infty\}$*.*

Our complexity analysis derives a recurrence relation for the complexity function by expressing the length of a trace in terms of the lengths of traces that start from lower rank states. This is achieved by

- attaching to each initial state a *rank* from a well-founded set that we use as the argument of the complexity function and that we recur over, and
- defining a *squeezer* that maps each state from the original trace to a state in a lower-rank trace; the mapping forms a *partitioned simulation* according to a *partition function* that decomposes a trace to segments; each segment is simulated

by a (separate) lower-rank trace, allowing to express the length of the former in terms of the latter, and finally,

- defining a *rank bounding function* that expresses (an upper bound on) the ranks of the lower-rank traces in terms of the rank of the higher-rank trace.

We elaborate on these components next.

5.3.1 Time complexity as a function of rank

We start by defining a rank function that allows us to express the time complexity of an initial state by means of its rank. This is to make this chapter self contained, and emphasize that in contrast to chapter 4, here the rank has a more important role - it should not only decrease, but the rank decrease should be well understood.

Definition 11 (Rank)**.** *Let X be a set, and $\prec$ be a well-founded partial order over X. Let $B \supseteq \min(X)$ be a* base *for X, where $\min(X)$ is the set of all the minimal elements of X w.r.t. $\prec$. A* rank function *$r : init \to X$ maps each initial state to a rank in X. We extend the notion of a rank to initial traces as follows. Given an initial trace $\tau = \tau(\sigma_0)$, we define its rank to be the rank of σ_0. We refer to states σ_0 such that $r(\sigma_0) \in B$ as the* base states*. Similarly, (initial) traces whose ranks are in B are called* base traces*.*

In our analysis, ranks range over $X = \mathbb{N}^m$ (for some $m \in \mathbb{N}^+$), with $\prec$ defined by the lexicographic order. Ranks let us abstract away data inside the initial execution states which does *not* affect the worst-case bound on the trace length. For example, the length of traces of the binary counter program (Figure 5.1) is completely agnostic to the actual content of the array at the initial state. The only parameter that affects its trace length is the array size, and not which integers are stored inside it. Hence, a suitable rank function in this example maps an initial state to its array length. This is despite the fact that the execution does depend on the content of the array, and, in particular, the number of remaining iterations from an intermediate state within the execution depends on it. The partial order $\prec$ and the base set B will be used to define the recurrence formula as we explain in the sequel.

We will assume from now on that $(X, \prec, B)$, as well as the rank function, are fixed, and can be understood from context. The rank function r induces a complexity function $comp_x : X \to \mathbb{N} \cup \{\infty\}$ over ranks, defined as follows.

Definition 12 (Complexity over ranks)**.** *The complexity function over ranks, $comp_x : X \to \mathbb{N} \cup \{\infty\}$, is defined by:*

$$comp_x(rank) = \max\{comp_s(\sigma_0) \mid r(\sigma_0) \preceq rank \wedge \sigma_0 \in init\}$$

The definition ensures that for every initial state $\sigma_0 \in init$, we can compute (an upper bound on) its time complexity based on its rank, as follows: $comp_s(\sigma_0) \leq comp_x(r(\sigma_0))$. The complexity of *rank* takes into account all states with $r(\sigma) \preceq rank$ and not only those with rank exactly *rank*, to ensure monotonicity of $comp_x$ in the rank (i.e., if $rank_1 \preceq rank_2$ then $comp_x(rank_1) \leq comp_x(rank_2)$). Our approach is targeted at extracting a recurrence relation for $comp_x$.

5.3.2 Complexity decomposition by partitioned simulation

In order to express the length of a trace in terms of the lengths of traces of lower ranks, we use a *squeezer* that maps states from the original trace to states of lower-rank traces and (implicitly) induces a correspondence between the original trace and the lower-rank trace(s). For now, we do not require the squeezer to decrease the rank of the trace; this requirement will be added later. The squeezer is accompanied by a partition function to form a *partitioned simulation* that allows a single higher-rank trace to be matched to multiple lower-rank traces such that their lengths may be correlated.

Definition 13 (Squeezer, $\curlyvee$)**.** *A squeezer is a function $\curlyvee : \Sigma \to \Sigma$.*

Definition 14. *A function $p_d : \Sigma \to \{1, \ldots, d\}$, where $d \in \mathbb{N}^+$ is called a d-*partition function *if for* every *trace $\tau = \sigma_0, \sigma_1, \ldots$ it holds that $p_d(\sigma_{i+1}) \geq p_d(\sigma_i)$ for every $0 \leq i < |\tau|$.*

The partition function partitions a trace into a bounded number of *segments*, where each segment consists of states with the same value of p_d. We refer to the first state of a segment as a *switch state*, and to the last state of a finite segment as a *last state* (note that if τ is infinite, its last segment has no last state). In particular, this means that the initial state of a trace is a switch state. (Note that a state may be a switch state in one trace but not in another, while a last state is a last state in any trace, as long as the same partition function is considered.)

Our complexity analysis requires the squeezer to form a partitioned simulation with respect to p_d. Roughly, this means that the squeezer maps each segment of a trace to a (lower-rank) trace that "simulates" it. To this end, we require *all* the states σ within a segment of a trace to be (h,ℓ)-"stuttering", for some $h \geq \ell \geq 1$. Stuttering lets h consecutive transitions of σ be matched to ℓ consecutive transitions of its squeezed counterpart. If $h = \ell$, the state σ contributes to the complexity the same number of steps as the squeezed state. Otherwise, σ contributes $h - \ell$ additional steps, resulting in a longer trace. Recall that terminal states also have outgoing transitions (to themselves), however these transitions do not capture actual steps; they do not contribute to the complexity. Hence, stuttering also requires that "real" transitions of σ are matched to "real" transitions of its squeezed counterpart, namely, if the latter encounter a terminal state, so must the former. For the last states of segments the requirement is slightly different as the simulation ends at the last state, and a new simulation begins in the next segment. In order to account for the transition from the last state of one segment to the first (switch) state of the next segment, last states are considered $(2,1)$-stuttering if they are squeezed into terminal states, unless they are terminal themselves[1]. In any other case, they are considered $(1,1)$-stuttering. The formal definitions follow.

Definition 15 (Stuttering States)**.** *A non-last state $\sigma \in \Sigma$ is called a (h,ℓ)*-stuttering *state, for $h \geq \ell \geq 1$, if:*

- $tr^{\ell}(\Upsilon(\sigma)) = \Upsilon(tr^{h}(\sigma))$*;*
- *for every $i < \ell$, $tr^{i}(\Upsilon(\sigma)) \notin F$;*
- $tr^{\ell}(\Upsilon(\sigma)) \in F$ *implies that* $\Upsilon(tr^{h}(\sigma)) \in F$*.*

A last state $\sigma \in \Sigma$ is $(1,1)$-stuttering *if $\sigma \in F$ or $\Upsilon(\sigma) \notin F$. Otherwise, it is $(2,1)$*-stuttering.

To obtain a partitioned simulation, switch states (along any trace), which start new segments, are further required to be squeezed into initial states (since our complexity analysis only applies to initial states). We denote by $\mathbb{S}_{p_d}(\tau)$ the switch states of trace τ according to partition p_d and by $\mathbb{S}_{p_d}$ the switch states of *all* traces according to the partition p_d. Namely, $\mathbb{S}_{p_d} = \mathit{init} \cup \big\{ tr(\sigma) \;\big|\; \sigma \in \mathit{reach} \wedge p_d(\sigma) < p_d\big(tr(\sigma)\big) \big\}$.

[1] Considering a non-terminal last state that is squeezed into a terminal state as $(1,0)$-stuttering may have been more intuitive than $(2,1)$-stuttering, but both properly capture the discrepancy between the number of transitions in the higher and lower rank traces, and $(2,1)$ better fits the rest of the technical development, which assumes that $h_i, \ell_i \geq 1$.

Definition 16 (Partitioned Simulation)**.** *We say that a squeezer $\curlyvee : \Sigma \to \Sigma$ forms a* $\{(h_i, \ell_i)\}_{i=1}^{n}$-partitioned simulation *according to p_d and denote $\curlyvee \sim \mathbb{PS}_{p_d}(\{(h_i, \ell_i)\}_{i=1}^{n})$ if for every reachable state σ we have that:*

- *σ is (h_i, ℓ_i)-stuttering for some $1 \le i \le n$, and*
- *$\sigma \in \mathbb{S}_{p_d} \Rightarrow \curlyvee(\sigma) \in init$.*

Note that Definition 15 implies that a non-terminal state may only be squeezed into a terminal state if it is the last state in its segments. When $\{(h_i, \ell_i)\}_{i=1}^{n}$ is irrelevant or clear from the context, we omit it from the notation and simply write $\curlyvee \sim \mathbb{PS}_{p_d}$.

A trace squeezed by $\curlyvee \sim \mathbb{PS}_{p_d}(\{(h_i, \ell_i)\}_{i=1}^{n})$ may have an unbounded number of (h_i, ℓ_i)-stuttering states, which hinders the ability to define a recurrence relation based on the simulation. To overcome this, our complexity decomposition may use $\widehat{k} \ge 1$ to capture a common multiplicative factor of *all* the stuttering pairs, with the target of leaving only a *bounded* number of states whose stuttering exceeds $\widehat{k}$ and needs to be added separately. This will become important in Theorem 5.

Observation 1 (Complexity decomposition)**.** *Let $\curlyvee \sim \mathbb{PS}_{p_d}(\{(h_i, \ell_i)\}_{i=1}^{n})$, and $\widehat{k} \ge 1$. Let $\mathbb{E}_{\widehat{k}} \subseteq \{1, \dots, n\}$ be the set of indices such that $\frac{h_i}{\ell_i} > \widehat{k}$. Then for every $\sigma_0 \in init$ we have that*

$$comp_s(\sigma_0) \le \sum_{\sigma \in \mathbb{S}_{p_d}(\tau(\sigma_0))} \widehat{k} \cdot comp_s(\curlyvee(\sigma)) + \sum_{i \in \mathbb{E}_{\widehat{k}}} \sum_{\sigma \in \mathbb{K}_i(\tau(\sigma_0))} h_i - \ell_i \cdot \widehat{k}$$

where $\mathbb{K}_i(\tau(\sigma_0))$ is the multiset of (h_i, ℓ_i)-stuttering states in $\tau(\sigma_0)$.

In the observation, the first addend summarizes the complexity contributed by all the lower-rank traces, while using $\widehat{k}$ as an upper bound on the "inflation" of the traces. However, the states that are (h_i, ℓ_i)-stuttering with $\frac{h_i}{\ell_i}$ that exceeds $\widehat{k}$ contribute additional $h_i - (\ell_i \cdot \widehat{k})$ steps to the complexity, and as a result, need to be taken into account separately. This is handled by the second addend, which adds the steps that were not accounted for by the first addend. While we use the same inflation factor $\widehat{k}$ across the entire trace, a simple extension of the decomposition property may consider a different factor $\widehat{k}$ in each segment. Note that the first addend always sums over a finite number of elements since the number of switch states is at most d – the number of segments. If $\tau(\sigma_0)$ is finite, the second addend also sums over a finite number of elements.

Observation 1 considers the complexity function over states, and is oblivious to the rank. In particular, it does not rely on the squeezer decreasing the rank of states. Next, we use this observation as the basis for extracting a recurrence relation for the complexity function over ranks, in which case, decreasing the rank becomes important.

5.3.3 Extraction of recurrence relations over ranks

Based on the complexity decomposition, we define recurrence relations that capture $comp_x$ — the time complexity of the initial states as a function of their ranks. To go from the complexity as a function of the actual states (as in Observation 1) to the complexity as a function of their ranks, we need to express the rank of $\curlyvee(\sigma_s)$ for a switch state σ_s as a function of the rank of σ_0. To this end, we define $\hat{\curlyvee}$:

Definition 17. *Given r, $\curlyvee$ and p_d such that $\curlyvee \sim \mathbb{PS}_{p_d}$, a function $\hat{\curlyvee} : X \times \{1, \dots, d\} \rightarrow X$ is a* rank bounding function *if for every $rank \in X - B$ and $1 \leq i \leq d$, if $\tau(\sigma_0)$ is an initial trace such that $r(\sigma_0) = rank$, and $\sigma_s \in \mathbb{S}_{p_d}(\tau(\sigma_0))$ is a switch state such that $p_d(\sigma_s) = i$, the following holds:*

$$\text{(i) upper bound: } r\big(\curlyvee(\sigma_s)\big) \preceq \hat{\curlyvee}(rank, i) \quad \text{and} \quad \text{(ii) rank decrease: } \hat{\curlyvee}(rank, i) \prec rank$$

In other words, Definition 17 requires that for every non-base initial state $\sigma_0 \in init$ and switch state σ_s at segment i of $\tau(\sigma_0)$, we have that $r(\curlyvee(\sigma_s)) \preceq \hat{\curlyvee}(r(\sigma_0), i) \prec r(\sigma_0)$. Recall that $r(\curlyvee(\sigma_s))$ is well defined since $\curlyvee(\sigma_s)$ is required to be an initial state. The definition states that $\hat{\curlyvee}(rank, i)$ provides an upper bound on the rank of squeezed switch states in a non-base trace of rank *rank*. Monotonicity of $comp_x$ ensures that $comp_x(r(\curlyvee(\sigma))) \leq comp_x(\hat{\curlyvee}(rank, i))$. This definition also requires the rank of non-base traces to strictly decrease when they are squeezed, as captured by the “rank decrease” inequality.

Obtaining a rank bounding function, or even verifying that a given $\hat{\curlyvee}$ satisfies this requirement, is a challenging task. We return to this question later in this section.

These conditions allow to substitute the states for ranks in the first addend of Observation 1, and hence obtain recurrence relations for $comp_x$ over the (decreasing) ranks. To handle the second addend, we also need to bound the number of states whose stuttering, $\frac{h_i}{\ell_i}$, exceeds $\widehat{k}$. This is summarized by the following theorem:

Theorem 5. *Let $r : init \to X$ be a rank function, $\curlyvee : \Sigma \to \Sigma$ a squeezer and $p_d : \Sigma \to \{1, \ldots, d\}$ a partition function such that $\curlyvee \sim \mathbb{PS}_{p_d}\big(\{(h_i, \ell_i)\}_{i=1}^{n}\big)$. Let $\hat{\curlyvee} : X \times \{1, \ldots, d\} \to X$ be a rank bounding function w.r.t. r, $\curlyvee$ and p_d. If, for some $\widehat{k} \geq 1$, the number of (h_i, ℓ_i)-stuttering states that appear along* any *non-base initial trace is bounded by a constant $b_i \in \mathbb{N}$ whenever $i \in \mathbb{E}_{\widehat{k}}$, then*

$$comp_x(rank) \leq \sum_{i=1}^{d} \widehat{k} \cdot comp_x\big(\hat{\curlyvee}(rank, i)\big) + \sum_{i \in \mathbb{E}_{\widehat{k}}} b_i \cdot \big(h_i - \ell_i \cdot \widehat{k}\,\big). \tag{5.6}$$

Note that a state may be (h_i, ℓ_i)-stuttering for several i's, in which case, it is sound to count it towards any of the b_i's; in particular, we choose the one that minimizes $h_i - \ell_i \cdot \widehat{k}$.

Corollary 1. *Under the premises of Theorem 5, if $f : X \to \mathbb{N} \cup \{\infty\}$ satisfies $f(rank) = \sum_{i=1}^{d} \widehat{k} \cdot f(\hat{\curlyvee}(rank, i)) + \sum_{i \in \mathbb{E}_{\widehat{k}}} b_i \cdot (h_i - \ell_i \cdot \widehat{k})$ for every $rank \in X - B$, and $comp_x(rank) \leq f(rank)$ for every $rank \in B$, then $comp_x(rank) \leq f(rank)$ for every $rank \in X$. We conclude that $comp_s(\sigma_0) \leq f(r(\sigma_0))$ for every $\sigma_0 \in init$.*

Base-case complexity In order to apply Corollary 1, we need to accompany Equation (5.6) with a bound on $comp_x(\rho)$ for the base ranks, $\rho \in B$. Fortunately, this is usually a significantly easier task. In particular, the running time of the base cases is often constant, because intuitively, the following are correlated:

- the rank,
- the size of the underlying data structure, and
- the number of iterations.

In this case, symbolic execution may be used to obtain bounds for base cases (as we do in our work). In essence, any method that can yield a closed-form expression for the complexity of the base cases is viable. In particular, we can apply our technique on the base case as a subproblem.

5.3.4 Establishing the requirements of the recurrence relations extraction

Theorem 5 defines a recurrence relation from which an upper bound on the complexity function, $comp_x$, can be computed (Corollary 1). However, to ensure correctness, the premises of Theorem 5 must be verified. The requirement that $\Upsilon \sim \mathbb{PS}_{p_d}(\{(h_i, \ell_i)\}_{i=1}^{n})$ (see Definition 16) may be verified *locally* by examining individual (reachable) states: for any (reachable) state σ, the check for (h_i, ℓ_i)-stuttering and switch states can, and should, be done in tandem, and require only observing at most $\max_i h_i$ transition steps from σ and $\max_i \ell_i$ from $\Upsilon(\sigma)$. In contrast, the property required of $\hat{\Upsilon}$ is *global*: it requires $\hat{\Upsilon}(rank, i)$ to provide an upper bound on the rank of *any* squeezed switch state that may occur in *any* position along *any* non-base initial trace whose initial state has rank *rank*. Similarly, the property required of the bounds b_i is also *global*: that the number of (h_i, ℓ_i)-stuttering states along *any* non-base initial trace is at most b_i. It is therefore not clear how these requirements may be verified in general. We overcome this difficulty by imposing additional restrictions, as we discuss next.

5.3.4.1 Establishing bounds on the number of occurrences of stuttering states

Bounds on the number of occurrences *per trace* that are sound *for every trace* are difficult to obtain in general. While clever analysis methods exist that can do this kind of accounting, we found that a stronger, simpler condition applies in many cases:

- For every $\sigma \in reach$, either:
- σ is (h_i, ℓ_i)-stuttering with $\frac{h_i}{\ell_i} \leq \widehat{k}$; <u>*or*</u>
- σ is (h_i, ℓ_i)-stuttering (with $\frac{h_i}{\ell_i} > \widehat{k}$), *and* either σ is a <u>switch state</u> or $tr^{h_i}(\sigma)$ is a <u>last state</u>.

This restricts these cases to occur only at the beginnings and ends of segments. It implies a total bound of $2d \cdot \max_i(h_i - \ell_i \cdot \widehat{k})$ on the "surplus" of any trace, therefore, we substitute this expression for the rightmost sum in Equation (5.6).

5.3.4.2 Validating a rank bounding function

The definition of a rank bounding function (Definition 17) encapsulates two parts. Part (ii) ensures that the rank decreases: $\hat{\Upsilon}(rank, i) \prec rank$ for every $rank \in X - B$. Verifying that this requirement holds does not involve any reasoning about the states, nor traces, of the transition system. Part (i) ensures that $\hat{\Upsilon}$ provides an upper bound on the rank of squeezed switch states. Formally, it requires that $r(\Upsilon(\sigma_s)) \preceq \hat{\Upsilon}(r(\sigma_0), i)$

for every switch state σ_s in segment $i \in \{1, \dots, d\}$ along a trace that starts from a non-base initial state σ_0. Namely, it relates the rank of the squeezed switch state, $\curlyvee(\sigma_s)$, to the rank of the initial state, σ_0, where no bound on the length of the trace between the initial state σ_0 and the switch state σ_s is known apriori. As such, it involves global reasoning about traces. We identify two cases in which such reasoning may be avoided:

- The partition p_d consists of a single segment (i.e., $d = 1$); or
- The rank function extends to *any* state (and not just the initial states), while being preserved by tr.

In both of these cases, we are able to verify the correctness of $\hat{\curlyvee}$ locally.

A single segment. In this case, the only switch state along a trace is the initial state, and hence the upper-bound requirement of $\hat{\curlyvee}$ boils down to the requirement that for every $\sigma_0 \in init$ such that $r(\sigma_0) \in X - B$, we have that $r(\curlyvee(\sigma_0)) \preceq \hat{\curlyvee}(r(\sigma_0), 1)$.

Lemma 5. *Let r, $\curlyvee$ and $p_1 : \Sigma \to \{1\}$ such that $\curlyvee \sim \mathbb{PS}_{p_1}$. Then $\hat{\curlyvee} : X \times \{1\} \to X$ satisfies the upper-bound requirement of a rank bounding function if and only if $r(\curlyvee(\sigma_0)) \preceq \hat{\curlyvee}(r(\sigma_0), 1)$ for every $\sigma_0 \in init$ such that $r(\sigma_0) \in X - B$.*

Rank preservation. Another case in which the upper-bound property of $\hat{\curlyvee}$ may be verified locally is when the r can be extended to *all* states while being preserved by tr:

Definition 18. *A function $\hat{r} : \Sigma \to X$* extends *the rank function $r : init \to \Sigma$ if $\hat{r}$ agrees with r on the initial states, i.e., $\hat{r}(\sigma_0) = r(\sigma_0)$ for every initial state $\sigma_0 \in init$. The extended rank function $\hat{r}$ is* preserved by *tr, if for every reachable state σ, we have that $\hat{r}(tr(\sigma)) = \hat{r}(\sigma)$.*

Preservation of $\hat{r}$ by tr ensures that all states along a (reachable) trace share the same rank. In particular, for a reachable switch state σ_s that lies along $\tau(\sigma_0)$, rank preservation ensures that $\hat{r}(\sigma_s) = \hat{r}(\sigma_0) = r(\sigma_0)$ (the last equality is due to the extension property), allowing us to recover the rank of σ_0 from the rank of σ_s. Therefore, the upper-bound requirement of $\hat{\curlyvee}$ simplifies into the *local* requirement that for every reachable switch state σ_s such that $\hat{r}(\sigma_s) \in X - B$, we have that $\hat{r}(\curlyvee(\sigma_s)) \preceq \hat{\curlyvee}(\hat{r}(\sigma_s), i)$, for every $i \in \{1, \dots, d\}$.

Lemma 6. *Let r, $\curlyvee$ and $p_d : \Sigma \to \{1, \dots, d\}$ such that $\curlyvee \sim \mathbb{PS}_{p_d}$. Suppose that $\hat{r} : \Sigma \to X$ extends r and is preserved by tr. Then $\hat{\curlyvee} : X \times \{1, \dots, d\} \to X$ satisfies the upper-bound requirement of a rank bounding function if and only if $\hat{r}(\curlyvee(\sigma_s)) \preceq \hat{\curlyvee}(\hat{r}(\sigma_s), i)$ for every reachable switch state σ_s such that $\hat{r}(\sigma_s) \in X - B$ and for every $i \in \{1, \dots, d\}$.*

Remark 1. *The notion of a partitioned simulation requires a switch state σ_s to be squeezed into an initial state. This requirement may be relaxed into the requirement that σ_s is squeezed into a* reachable *state $\curlyvee(\sigma_s)$, provided that we are able to still ensure that the rank of (some)* initial *state σ_0' leading to $\curlyvee(\sigma_s)$ is smaller than the rank of the trace on which σ_s lies, and that the rank of σ_0' is properly captured by $\hat{\curlyvee}$. One case in which this is possible, is when r is extended to $\hat{r}$ that is preserved by tr, as in this case $\hat{r}(\curlyvee(\sigma_s)) = \hat{r}(\sigma_0') = r(\sigma_0')$.*

This subsection described *local* properties that ensure that a given program satisfies the requirements of Theorem 5. The locality of the properties facilitates the use of SMT solvers to perform these checks automatically. This is a key step for effective application of the method.

5.3.5 Trace-length vs. state-size recurrences with squeezers

A plethora of work exists for analyzing the complexity of programs (see Section 5.6 for a discussion of related works). Most existing techniques for automatic complexity analysis aim to find a recurrence relation on the length of the execution trace, relating the length of a trace from some state to the length of the remaining trace starting at its successor. These are recurrences on *time*, if you will, whereas our approach generates recurrences on the state *size* (captured by the rank). Is our approach completely orthogonal to preceding methods? Not quite. It turns out that from a conceptual point of view, our approach can formulate a recurrence on time as well, as we demonstrate in this section.

Obtaining trace-length recurrences based on state squeezers The key idea is to use tr itself as a squeezer that squeezes each state into its immediate successor. Putting aside the initial-anchor requirement momentarily, such a squeezer forms a partitioned simulation with a single segment (i.e., $p_d \equiv 1$), in which all the states along a trace are $(1, 1)$-stuttering, except for the last one (if the trace is finite), which is $(2, 1)$-stuttering. Recall that squeezers must also preserve initial states (see Definition 16), a property that may be violated when $\curlyvee = tr$, as the successor of an initial state is not necessarily an initial state. We restore the initial-anchor property by setting $\widehat{init} = \Sigma$, i.e., every state is considered an initial state[2].

A consequence of this definition is that $comp_x$ will now provide an upper bound on the time complexity of *every* state and not only of the initial states, in terms of a rank that needs to be defined. If we further define a rank-bounding function $\hat{\curlyvee}$ we may extract a recurrence relation of the form

$$comp_x(rank) = comp_x(\hat{\curlyvee}(rank)) + 1$$

(we use $\hat{\curlyvee}(rank)$ as an abbreviation of $\hat{\curlyvee}(rank, 1)$, since this is a special case where $d = 1$).

Defining the rank and the rank bounding function Recall that the rank $r : \Sigma \to X$ captures the features of the (initial) states that determine the complexity. To allow maximal precision, especially since *all* states are now initial, we set X to be the set of *states* Σ, and define r to be the identity function, $r(\sigma) = \sigma$. With this definition, $comp_x$ and $comp_s$ become one. Next, we need to define $\prec$ and B, while ensuring that $\curlyvee$ squeezes the (non-base) initial states, which are now *all* the states, into states of a lower rank according to $\prec$. Since squeezers act like transitions now, having that $\curlyvee = tr$, they have the effect of decreasing the number of transitions remaining to reach a terminal state (provided that the trace is finite). We use this observation to define $\prec \subseteq \Sigma \times \Sigma$. Care is needed to ensure that $(\Sigma, \prec)$ is well-founded, i.e., every descending chain is finite, even though the program may *not* terminate. Here is the definition that achieves this goal:

$$\sigma_1 \prec \sigma_2 \;\Leftrightarrow\; comp_s(\sigma_1) < comp_s(\sigma_2) \tag{5.7}$$

Since $\curlyvee = tr$ does not decrease $comp_s$ for states that belong to infinite (non-terminating) traces ($comp_s(\curlyvee(\sigma)) = comp_s(\sigma) = \infty$, hence $\curlyvee(\sigma) \not\prec \sigma$), they must be included in B, together with the terminal states, which are minimal w.r.t. $\prec$. Namely, $B = F \cup \{\sigma \mid comp_s(\sigma) = \infty\}$. Technically, this means that the base of the recurrence needs to define $comp_x$ for these states.

The final piece in the puzzle is setting $\hat{\curlyvee} = tr$. Since $\curlyvee \sim \mathbb{PS}_{pd}\big(\{(1,1),(2,1)\}\big)$

[2]In fact, it suffices to consider $\widehat{init} = reach$, in which case we may be able to take advantage of information from static analyses

(when $\widehat{init} = \Sigma$), where the number of $(2,1)$-stuttering states that appear along any non-base initial trace is bounded by 1, we may use Theorem 5, setting $\widehat{k} = 1$, to derive the following recurrence relation, which reflects induction over time:

$$comp_x(\sigma) = comp_x(tr(\sigma)) + 1.$$

The formulation above represents a degenerate, naïve, choice of ingredients for the sake of a theoretical construction, whose purpose is to lay the foundation for a general framework that takes its strengths from both induction over time and induction over rank. This construction does not exploit the full flexibility of our framework. In particular, ranking functions obtained from termination proofs, as used in [8], may be used to augment the rank in this setting. Further, invariants inferred from static analysis can be used to refine the recurrences.

5.4 Synthesis

So far we have assumed that the rank function r, partition function p_d, squeezer $\curlyvee$ and a rank bounding function $\hat{\curlyvee}$ are all readily available. Clearly, they are specific to a given program. It would be too tedious for a programmer to provide these functions for the analysis of the underlying complexity. In this section we show how to automate the process of obtaining $(r, p_d, \curlyvee, \hat{\curlyvee})$ for a class of typical looping programs. We take advantage of the fact that these components are much more compact than other kinds of auxiliary functions commonly used for resource analysis, such as monotonically decreasing measures used as ranking functions. For example, a ranking function for the binary counter program shown in Figure 5.1 is:

$$m(n,i,c) = \left(n \cdot \sum_{j=0}^{n-1} 2^j \cdot c[j]\right) + (2^i - 1) + (n - i)$$

whereas the rank, partition, $\curlyvee$ and $\hat{\curlyvee}$ are

$$\begin{aligned} r(n,i,c) &= n & \curlyvee(n,i,c) &= \big(n-1, (i \geq n)\ ?\ i-1 : i, c[:n-1]\big) \\ \hat{\curlyvee}(rank) &= rank - 1 & p_d(n,i,c) &= (i \geq n \mid\mid c[n-1])\ ?\ 2 : 1 \end{aligned}$$

This enables the use of a relatively naïve enumerative approach of multi-phase generate-and-test, employing some early pruning to discard obviously non-qualifying candidates.

5.4.1 SyGuS

The generation step of the synthesis loop applies syntax guided synthesis (SyGuS [12]). Like any other SyGuS method, defining the underlying grammars is more art than science. It should be expressive enough to capture the desired terms, but strict enough to effectively bound the search space.

Ranks are taken from $\mathbb{N}^m$ where $m \in \{1, 2, 3\}$ and $\prec$ is the usual lexicographic order. The rank function r comprises of one expression for each coordinate, constructed by adding / subtracting integer variables and array sizes. Boolean variables are not used in rank expressions.

Partition functions p_d. Our implementation currently supports a maximum number of two segments. This means that the partition function only assigns the values 1 and 2, and we synthesize it by generating a condition over the program's variables, cond, that selects between them: $p_d(\sigma) = \mathit{cond}(\sigma)\ ?\ 2 : 1$. Handling up to two segments is *not* an inherent limitation, but we found that for typically occurring programs, two segments are sufficient.

Squeezers $\curlyvee$ are the only ingredient that requires substantial synthesis effort. We represent squeezers as small loop-free imperative programs, which are natural for representing state transformations. We use a rather standard syntax with 'if-then-else' and assignments, plus a `remove-adjust` operation that removes array entries and adjusts indices relating to them accordingly. .

Rank bounding functions $\hat{\curlyvee}$. With a well-chosen squeezer $\curlyvee$, it suffices to consider quite simple rank bounds for the mini-traces. Hence, the rank-bounds defined by $\hat{\curlyvee}$ are obtained by adding, subtracting and multiplying variables with small constants (for each coordinate of the rank). Similar to the choice of ranks, targeting simple expressions for $\hat{\curlyvee}$ helps reduce the complexity of the final recurrence that is generated from the process.

5.4.2 Verification

For the sake of verifying the synthesized ingredients, we fix a set $\{h_i, \ell_i\}$ of stuttering shapes, and check the requirements of Theorem 5 as discussed in Section 5.3.4. In particular, we check that p_d is weakly monotone, i.e., that *cond* cannot change from true to false in any step of tr. Note that some of the properties may be used to discriminate some of the ingredients independent of the others. For example, the simulation requirement only depends on Υ and p_d.

Unbounded verification Once candidates pass a preliminary screening phase, they are verified by encoding the program and all the components $r, p_d, \Upsilon, \hat{\Upsilon}$ as first-order logic expressions, and using an SMT solver (Z3 [39]) to verify that the requirements are fulfilled for all traces of the program.

As mentioned in Section 5.3.4, all the checks are local and require observing a bounded set of steps starting from a given σ. The only facet of the criteria that is difficult to encode is the fact they are required of the reachable states (and not any state). Of course, if we are able to ascertain that these are met for *all* $\sigma \in \Sigma$, including unreachable states, then the result is sound. However, for some programs and squeezers, the required properties (esp., simulation) do not hold universally, but are violated by unreachable states. To cope with this situation without having to manually provide invariants that capture properties of the reachable states, we use a CHC solver, Spacer [90], which is part of Z3, to check whether all the reachable states in the unbounded-state system induced by the input program satisfy these properties. This can be seen as a reduction from the problem of verifying the premises of Theorem 5 to that of verifying a safety property.

5.5 Empirical Evaluation

We implemented our complexity analyzer as a publicly available tool, SqzComp, that receives a program in a subset of C and produces recurrence relations. SqzComp is written in C++, using the Z3 C++ API [39], and using Spacer [90] via its SMTLIB2-compatible interface. Since our squeezers may remove elements from arrays, we initially encoded arrays as SMT sequences. However, we found that it is beneficial to restrict squeezers to only remove the first or last elements of an array, resulting in a more efficient encoding with the theory of arrays. For the base case of generated recurrences, we use the symbolic execution engine KLEE [27] to bound the total number of iterations by a constant.

Description	Real	Inferred bound		SqzComp	
	complexity	CoFloCo	SqzComp	Time	d
array: max value	$O(\|A\|)$	$O(\|A\|)$	$O(\|A\|)$	$<$ 1 sec	1
array: min value	$O(\|A\|)$	$O(\|A\|)$	$O(\|A\|)$	$<$ 1 sec	1
array: find first	$O(\|A\|)$	$O(\|A\|)$	$O(\|A\|)$	$<$ 1 sec	1
array: find last	$O(\|A\|)$	$O(\|A\|)$	$O(\|A\|)$	$<$ 1 sec	1
array: is-sorted	$O(\|A\|)$	$O(\|A\|)$	$O(\|A\|)$	$<$ 1 sec	1
array: longest asc. prefix	$O(\|A\|)$	$O(\|A\|)$	$O(\|A\|)$	$<$ 1 sec	1
array: binary search	$O(\log(\|A\|))$	$O(\log(\|A\|))$	$O(\log(\|A\|))$	$<$ 1 sec	1
gcd	$\max(x, y)$	$O(x + y)$	$O(x + y)$	$<$ 1 sec	1
two-phase loop 1	$O(2n - 2x + y)$	$O(2n - 2x + y)$	$O(2n + 2y)$	$<$ 1 sec	1
two-phase loop 2	$O(n - x + m - y)$	$O(n - x + m - y)$	$O(n - x + m - y)$	$<$ 1 sec	1
two-phase loop 3	$O(n)$	$O(n)$	$O(n)$	$<$ 1 sec	1
two-phase loop 4	$O(2n - x - z)$	$O(2n - x - z)$	$O(2n)$	$<$ 1 sec	1
multi-path loop 1	$O(n)$	$O(3n)$	$O(n)$	$<$ 1 sec	1
multi-path loop 2	$O(n)$	$O(n)$	$O(n)$	$<$ 1 sec	1
multi-path loop 3	$O(n)$	$O(n)$	$O(n)$	$<$ 1 sec	1
tricky init loop	$O(z)$	$O(z)$	$O(z)$	4 min	1
nested loop 1	$O(\|x - y\|)$	$O(\|x - y\|)$	$O(x + y)$	$<$ 1 sec	1
nested loop 2	$O(a^2)$	$O(a^2)$	$O(a^2)$	16 min	1
context sensitive loop	$O(\max(n - m, m))$	$O(\max(n - m, m))$	$O(n)$	7 min	1
binary counter	$O(2^{n+1})$	∞	$O(2^{n+1})$	34 min	2
subsets	$O(\binom{n-m}{k})$	∞	$O(\binom{n-m}{k})$	50 min	2
monotone sequences	$O(\binom{n}{k})$	∞	$O(\binom{n}{k})$	50 min	2

Table 5.1: Experimental results. In array programs, A denotes an array. x, y, z, n, m, k, a are integer variables.

5.5.1 Experiments

We evaluated our tool, SqzComp, on a variety of benchmark programs taken from [49], as well as three additional programs: the binary counter example from Section 5.2, a subsets example, described in Section 5.5.2, and an example computing monotone sequences. These examples exhibit intricate time complexities. From the benchmark suite of [49] we filtered out non-deterministic programs, as well as programs that failed syntactic constraints that our frontend cannot currently handle. We compared SqzComp to CoFloCo [49]—the state of the art tool for complexity analysis of imperative programs.

Table 5.1 summarizes the results of our experiments. The first column presents the name of the program, which describes its characteristics (each of the "two-phase loop" programs consists of a loop with an if statement, where the branch executed changes starting from some iteration). The second column specifies the real complexity, while the following two columns present the bounds inferred by SqzComp and by CoFloCo, respectively. (For SqzComp, the reported bounds are the solutions of the recurrences

```
1 void subsets(uint n, uint k, uint m) {
2   uint I[k]; int j = 0; bool f = true;
3   while (j >= 0) {
4     if (j >= k) /*start left scan*/{f=false; j--;}
5     else if (j==0 && f) /*init*/{f=true;I[0]=m;j++;}
6     else if (f) /*right fill*/{f=true;I[j]=I[j-1]+1;j++;}
7     else if (I[j]>=n-k+j)/*left scan*/{f=false; j--;}
8     else /*start right fill*/{f=true; I[j]=I[j]+1;j++;}
9  }}
```

```
squeezer(uint I[], uint n, uint k, uint m, int j, bool f) {
  if       (I[0]==m && j>0) { m++; remove I[0]; k--; j--; }
  else if (I[0]==m)         { m++; remove I[0]; k--;      }
  else                      { m++;                        }
}
```

Figure 5.4: An example program that produces all subsets of $\{m, \ldots, n-1\}$ of size k; below is the synthesized squeezer.

output by the tool.) The fourth and fifth columns present the analysis running time, respectively the number of segments used in the analysis, of SqzComp.

CoFloCo's analysis time is always in the order of magnitude of 0.1 second, whether it succeeds to find a complexity bound or not. Our analysis is considerably slower, mostly due to the naïve implementation of the synthesizer. When both CoFloCo and SqzComp succeed, the bounds inferred by CoFloCo are sometimes tighter.

However, SqzComp manages to find tight complexity bounds for the new examples, which are not solved by CoFloCo, and to the best of our knowledge, are beyond reach of existing tools. (We also encoded the new examples as OCaml programs and ran the tool of [74] on them, and it failed to infer bounds.)

5.5.2 Case study: Subsets example

This subsection presents the Subsets example—one challenging example from our benchmarks— and the details of its complexity analysis. Notably, our method is able to infer a binomial bound, which is asymptotically tight.

The code, shown in Figure 5.4, iterates over all the subsets of {`m,...,n-1`} of size `k`. The "current" subset is maintained in an array `I` whose length is `k`, and which is always sorted, thus avoiding generating the same set more than once. The first k iterations of the loop fill the array with values {`m,m+1,...,m+k-1`}, which represent the first subset generated. This is taken care of by the branches at lines 5, 6 that perform a "right fill" phase, filling in the array with an ascending sequence starting from `m` at `I[0]`. Once the first k iterations are done, `j` reaches the end of the array (`j=k`) and so the next iteration will execute line 4, turning off the flag `f`, signifying that the array should now be scanned leftwards. In each successive iteration, `j` is decreased, looking for the rightmost element that can be incremented. For example, if $n = 8, I = [2, 6, 7]$, this rightmost element is $I[0] = 2$. After that element is incremented, the flag `f` is turned on again, completing the "left scan" phase and starting a "right fill" phase.

A univariate recurrence Consider the rank function $r(I, n, k, m, j, f) = n - m$, defined with respect to $(\mathbb{N}, <)$, and the squeezer shown below the program in Figure 5.4. The squeezer observes the first element of the array: if it is equal to m (the lower bound of the range), it removes it from the array, shrinking its size (k) by one. It then adjusts the index j to keep pointing to the same element; unless $j = 0$, in which case that element is removed. This squeezer forms a 2-partitioned simulation, as illustrated by the traces in Figure 5.5. All states are $(1, 1)$-stuttering, except for σ_0, which is $(2, 1)$-stuttering, as caused by the removal of $I[0]$ when $j = 0$. The rank bounding function is $\hat{\Upsilon}(i, rank) = rank - 1$ for $i \in \{1, 2\}$. We therefore obtain the following recurrence relation:

$$comp_x(rank) \leq 1 + comp_x(rank - 1) + comp_x(rank - 1).$$

The base of the recurrence is $comp_x(0) = 1$, leading to the solution $comp_x(rank) \leq 2^{rank+1} - 1$. This means that for an initial state, $comp_s(I, n, k, m, 0, \mathsf{true}) \leq comp_x(n - m) \leq 2^{n-m+1} - 1$.

A multivariate recurrence Consider an alternative rank definition $r(I, n, k, m, j, f) = (n - m, k)$ defined with respect to $(\mathbb{N} \times \mathbb{N}, <)$, where '$<$' denotes the lexicographic order, together with the same squeezer and partition as before. The rank bounding function is now $\hat{\Upsilon}\big((rank_1, rank_2), i\big) = \begin{cases} (rank_1 - 1, rank_2 - 1) & i = 1 \\ (rank_1 - 1, rank_2) & i = 2 \end{cases}$. The corresponding recurrence relation is:

$$comp_x(rank_1, rank_2) \leq 1 + comp_x(rank_1 - 1, rank_2 - 1) + comp_x(rank_1 - 1, rank_2)$$

with base $comp_x(0, \_) = 1$, resulting in the solution $comp_x(rank_1, rank_2) \leq \binom{rank_1+2}{rank_2}$. That is, for an initial state, $comp_s(I, n, k, m, 0, \mathsf{true}) \leq comp_x(n - m, k) \leq \binom{n-m+2}{k}$.

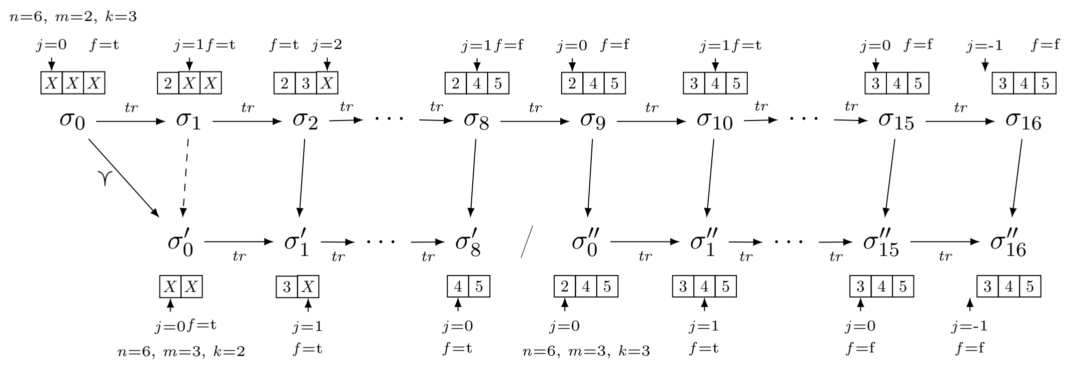


Figure 5.5: An illustration of the 2-partitioned simulation for the subsets example. In the univariate case, the rank of the upper trace is $n - m$ and that of the lower traces is $n - m - 1$. In the multivariate case, the upper trace is of rank $(n - m, k)$, lower traces of ranks $(n - m - 1, k - 1)$, $(n - m - 1, k)$.

Interestingly, this example demonstrates that the same squeezer may yield different recurrences, when different ranks (and rank bounding functions) are considered. It also demonstrates a case where different segments of a trace are mapped to mini-traces of a different rank.

5.6 Related Work

This section focuses on exploring existing methods for *static* complexity analysis of *imperative* programs. Dynamic profiling and analysis [100] are a separate research area, more related to testing, and generally do not provide formal guarantees. We further focus on works that determine *asymptotic* complexity bounds, and use the number of iterations executed as their cost model; we refrain from thoroughly covering previous techniques that analyze complexity at the instruction level.

Static cost analysis The seminal work of [149] defined a two steps meta-framework where recurrence relations are extracted from the underlying program, and then analyzed to provide closed-form upper bounds. Broadly speaking, cost relations are a generalized framework that captures the essence of most of the works mentioned in this section.

[6] and [49] infer cost relations of imperative programs written in Java and C respectively. Cost relations resemble somewhat limited C procedures: They are capable of recursive calls to other cost relations, and they can handle non-determinism that arises

either as a consequence of direct `nondet()` in the program, or as a result of inherent imprecision of static analysis. They define for every basic block of the program its own cost relation function, and then form chains according to the control flow graph of the program. They use numerical abstract domains to support a context sensitive analysis of whether a chain of visits to specific basic blocks is feasible or not. Once all infeasible chains are removed, disjunctive analysis determines an overall approximation of the heaviest chain, representing the max number of iterations.

[65] uses multiple counter instrumentation that are automatically inserted in various points in the code, initialized and incremented. These ghost counters enable to infer an overall complexity bound by applying appropriate abstract interpretation handling numeric domains. [66] and [61] apply code transformations to represent multi-path loops and nested loops in a canonical way. Then, paths connecting pairs of "interesting" code points π_1, π_2 (loop headers etc.) are identified, in a way that satisfies some properties. For instance, π_1 is reached twice *without* reaching π_2. The path property induces progress invariants, which are then analyzed to infer the overall complexity bound.

[97] define an abstraction of the program to a *size-change-graph*, where transition edges of the control flow graph are annotated to capture sound over-approximation relations between integer variables. The graph is then searched for infinitely decreasing sequences, represented as words in an ω-regular language. This representation concisely characterizes program termination. [156] then harnesses the size-change abstraction from [97] to analyze the complexity of imperative programs. First, they apply standard program transformations like pathwise analysis to summarize inner nested loops. Then, they heuristically define a set of scalar rank functions they call norms. These norms are somewhat similar to our rank function in the sense that they help to abstract away program parts that do not effect its complexity. The program is then represented as a size-change graph, and multi-path contextualization [98] prunes subsequent transitions which are infeasible.

[17] introduces *difference constraints* in the context of termination, to bound variables x' in current iteration with some y in previous iteration plus some constant c: $x' \leq y + c$. [132] extends difference constraints to complexity analysis. Indeed, it is quite often the case that ideas from the area of program termination are assimilated in the context of complexity analysis and vice versa. They exploit the observation

that typical operations on loop counters like increment, decrement and resets are essentially expressible as difference constraints. They design an abstraction based on the domain of difference constraints, and obtain relevant invariants which are then used in determining upper bounds. [23] is very similar, only that it represents a program as an integer transition system and allows nonlinear numerical constraints and ranking functions.

As we mentioned earlier, all of these approaches are based on identifying the progress of executions over time, characterizing the progress between two given points in the program. In contrast, our approach allows to reason over state size and compares whole executions.

Squeezers. The notion of squeezers was introduced by [78] for the sake of safety verification. As discussed in Section 5.1, the challenges in complexity analysis are different, and require additional ingredients beyond squeezers. [2, 3, 46] introduce *well structured transition systems*, where a well-quasi order (wqo) on the set of states induces a simulation relation. This property ensures decidability of safety verification of such systems (via a backward reachability algorithm). Our use of squeezers that decrease the rank of a state and induce a sort of a simulation relation may resemble the wqo of a well structured transition system. However, there are several key differences: we do not require the order (which is defined on ranks) to be a wqo. Further, we do not require a simulation relation between *any* states whose ranks are ordered, only between a state and its squeezed counterpart. Notably, our work considers complexity analysis rather than safety verification.

Chapter 6

Discussion

Automatic inversion of programs was first studied by Dijkstra who manually inverted simple array-manipulating programs [41]. Follow up works looked at inverting (i) simple programs whose semantics is given as logic programs [120], (ii) tree-traversal programs using relational calculus and deductive methods [29, 128], (iii) array transformers using techniques based on LR-parsing [55, 87] or testing [84], and (iv) bijective string-manipulating procedures [75, 101]. To the best of our knowledge, we are the first to apply machine learning tools to invert programs. We also note that the programs we invert are not necessarily injective. Recent advances in machine learning lead researchers to explore its capabilities in helping challenging program analysis tasks, e.g., specification inference [118, 124], speed up abstraction refinement [60], invariant generation [52, 108, 123], setting up parameters for parametrized static analyses [110], and infer clustering of variables in partially relational static analyses [70]. In our work, we address a dual question–how can machine learning technique help program analyses. To the best of our knowledge the question has not been widely addressed, with the notable exception of [107] which also argues that a combination of machine learning and program analysis can be a win-win situation. Another active research area is the use of input/output examples to learn computer programs. Often, this is done in the context of synthesis, where examples guide a search-based synthesis process [63]. For example, in [62], a learning procedure is used to synthesize string manipulating procedures which appears in the context of spreadsheets based on syntactic manipulation. Another attack on this problem was taken in [131], where the procedures were synthesized using database-like lookup operations. In these works, the focus is on designing a language in which programs can be synthesized and an efficient search heuristics. In this work we too focus on string manipulating procedures (SMPs), which are abundant in almost all software packages. However, instead of asking the user for input/output examples, we analyze the code of one procedure and its behavior, as expressed by input-output pairs, to synthesize another procedure. In [152], the authors suggest to learn the behavior of a procedure by inspecting its code and input-output examples. Their technique applies to a class of procedures which accepts their input character by character, e.g., multi-digit addition. They use recurrent neural network models with long short-term memory to accurately learn a model of the procedure behavior as a sequence-to-sequence transformer [141]. It can be interesting to see if a preliminary phase of program analysis, as we do in this work, can help improve the accuracy of their technique. String solvers, e.g., [16, 39, 88], can reason about constraints involving operations on strings. For example, HAMPI [88], can reason about constraints expressing membership in regular languages and fixed-size context-free languages. In contrast, we provide a technique based on a combination of machine learning and static analysis that can help invert string manipulating procedures written in a restricted programming language. Similar to our work, S-Looper [150] automatically summarises loops with the aim of improving program analysis. Their technique uses static analysis to enhance buffer-overflow detection. Our work is more general in that it is applicable to any analysis that operates on C directly, generating human-readable summaries that can even be used for refactoring. Godefroid and Luchaup [56] use partial loop summarisation to enable concolic execution to reason about multiple paths through a loop at once. Their summaries consist of pre- and post-conditions, which they automatically infer during concolic execution. Similarly, *loop-extended symbolic-execution* [126] uses a combination of symbolic execution and static analysis to summarise loops in order to speed up symbolic execution. As for S-Looper, these two approaches are intertwined with their analysis, unlike our approach which can be immediately used in any technique. STOKE [127] is an assembly level superoptimizer that speeds up loop-free code segments. With its recent extension to loops [31] their work is similar in spirit. They also use bounded verification to aid synthesis, but instead of a small-world theorem they use a sound verifier to generalise to arbitrary bounds.Srivastava et al. [136] present an approach synthesising loops from pre- and post-conditions using a verifier. While more precise, they require user-specified annotations, making it inapplicable as an automatic summarisation technique. LLVM's LoopIdiomRecognize pass attempts to replace loops that match `memset` or `memcpy` patterns and is quite specific to these functions (other compilers, such as GCC, have similar passes that recognise patterns). It detects induction variables from which it can recognise stride load and store instructions. Their to-do includes functions like `strlen` for over 6 years, showing that such passes require significant expertise to implement. By contrast, our approach is more general and can easily be extended. Program equivalence may be considered one of the most important problems in formal verification and has been the subject of decades of research [140]. Due to the vast literature on the topic and space, we only briefly review the subject. Proving program equivalence is useful in many domains ranging from translation validation [93, 105, 113, 122, 130], regression verification [57, 58], automatic merging [135], semantic differencing [38], and cross-version verification [69, 95]. One common approach for attacking the problem, e.g., [148], is establishing a simulation invariant between the states of two programs. Tracking the simulation enables defining a so-called correlating semantics which allows reasoning about correlated (interleaved) execution of two programs [14, 38, 142]. In contrast to these techniques, our approach focuses on establishing the equivalence of programs without co-executing them, but instead examines their input/output behaviour on bounded examples using symbolic execution. Symbolic execution-based methods [27, 30, 32, 33, 112, 117] often focus on practical equivalence verification up to a certain input bound. In contrast, we speculatively search for a synthesised program that agrees with the investigated loop on bounded inputs, and develop a small model theorem [115] which allows us to lift symbolic execution validated bounded equivalence to full equivalence. At the current state of affairs in automatic software verification of infinite state systems, the scene is dominated by various approaches with a common aim: computing over-approximations of unbounded executions by means of inferring loop invariants. Indeed, *abstract interpretation* [35], *property-directed reachability* [21], unbounded model checking [99], or template-based verification [138] can be seen as different techniques for computing such approximations by finding inductive loop invariants which are tight enough not to intersect with the set of bad behaviors. Experience has shown that these invariants are frequently quite hard to come by, even for seemingly simple and innocuous program, both automatically and manually. The purpose of this chapter is to suggest an alternative kind of correctness witness, which may be more amenable to automated search. We successfully applied our novel verification technique to array programs and managed to prove programs and properties which are beyond the ability of existing automatic verifiers. We believe that our approach can be combined with standard techniques to give rise to a new kind of hybrid techniques, where, e.g., a *partial* loop invariant is used as a baseline — verified via standard techniques — and is then *strengthened* to the desired safety property via squeezer-based verification. Dynamic profiling and analysis [100] are a separate research area, more related to testing, and generally do not provide formal guarantees. We further focus on works that determine *asymptotic* complexity bounds, and use the number of iterations executed as their cost model; we refrain from thoroughly covering previous techniques that analyze complexity at the instruction level. The seminal work of [149] defined a two steps meta-framework where recurrence relations are extracted from the underlying program, and then analyzed to provide closed-form upper bounds. Broadly speaking, cost relations are a generalized framework that captures the essence of most of the works mentioned in this section. [6] and [49] infer cost relations of imperative programs written in Java and C respectively. Cost relations resemble somewhat limited C procedures: They are capable of recursive calls to other cost relations, and they can handle non-determinism that arises either as a consequence of direct `nondet()` in the program, or as a result of inherent imprecision of static analysis. They define for every basic block of the program its own cost relation function, and then form chains according to the control flow graph of the program. They use numerical abstract domains to support a context sensitive analysis of whether a chain of visits to specific basic blocks is feasible or not. Once all infeasible chains are removed, disjunctive analysis determines an overall approximation of the heaviest chain, representing the max number of iterations. [65] uses multiple counter instrumentation that are automatically inserted in various points in the code, initialized and incremented. These ghost counters enable to infer an overall complexity bound by applying appropriate abstract interpretation handling numeric domains. [66] and [61] apply code transformations to represent multi-path loops and nested loops in a canonical way. Then, paths connecting pairs of “interesting” code points π_1, π_2 (loop headers etc.) are identified, in a way that satisfies some properties. For instance, π_1 is reached twice *without* reaching π_2. The path property induces progress invariants, which are then analyzed to infer the overall complexity bound. [97] define an abstraction of the program to a *size-change-graph*, where transition edges of the control flow graph are annotated to capture sound over-approximation relations between integer variables. The graph is then searched for infinitely decreasing sequences, represented as words in an ω-regular language. This representation concisely characterizes program termination. [156] then harnesses the size-change abstraction from [97] to analyze the complexity of imperative programs. First, they apply standard program transformations like pathwise analysis to summarize inner nested loops. Then, they heuristically define a set of scalar rank functions they call norms. These norms are somewhat similar to our rank function in the sense that they help to abstract away program parts that do not effect its complexity. The program is then represented as a size-change graph, and multi-path contextualization [98] prunes subsequent transitions which are infeasible. [17] introduces *difference constraints* in the context of termination, to bound variables x' in current iteration with some y in previous iteration plus some constant c: $x' \leq y + c$. [132] extends difference constraints to complexity analysis. Indeed, it is quite often the case that ideas from the area of program termination are assimilated in the context of complexity analysis and vice versa. They exploit the observation that typical operations on loop counters like increment, decrement and resets are essentially expressible as difference constraints. They design an abstraction based on the domain of difference constraints, and obtain relevant invariants which are then used in determining upper bounds. [23] is very similar, only that it represents a program as an integer transition system and allows nonlinear numerical constraints and ranking functions. As we mentioned earlier, all of these approaches are based on identifying the progress of executions over time, characterizing the progress between two given points in the program. In contrast, our approach allows to reason over state size and compares whole executions. The notion of squeezers was introduced by [78] for the sake of safety verification. As discussed in Section 5.1, the challenges in complexity analysis are different, and require additional ingredients beyond squeezers. [2, 3, 46] introduce *well structured transition systems*, where a well-quasi order (wqo) on the set of states induces a simulation relation. This property ensures decidability of safety verification of such systems (via a backward reachability algorithm). Our use of squeezers that decrease the rank of a state and induce a sort of a simulation relation may resemble the wqo of a well structured transition system. However, there are several key differences: we do not require the order (which is defined on ranks) to be a wqo. Further, we do not require a simulation relation between *any* states whose ranks are ordered, only between a state and its squeezed counterpart. Notably, our work considers complexity analysis rather than safety verification.

Bibliography

[1] http://www.dataparksearch.org/. 39

[2] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 313–321, 1996. doi: 10.1109/LICS.1996.561359. URL `https://doi.org/10.1109/LICS.1996.561359`. 77, 114, 119

[3] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160 (1-2):109–127, 2000. doi: 10.1006/inco.1999.2843. URL `https://doi.org/10.1006/inco.1999.2843`. 77, 114, 119

[4] Parosh Aziz Abdulla, Frédéric Haziza, and Lukáš Holík. All for the price of few. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 476–495, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 78

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006. ISBN 0321486811. 44

[6] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, pages 113–132, 2007. 86, 112, 118

[7] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic

inference of upper bounds for recurrence relations in cost analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis*, pages 221–237, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-69166-2. 82, 94

[8] Elvira Albert, Miquel Bofill, Cristina Borralleras, Enrique Martin-Martin, and Albert Rubio. Resource analysis driven by (conditional) termination proofs. *Theory Pract. Log. Program.*, 19(5-6):722–739, 2019. doi: 10.1017/S1471068419000152. URL https://doi.org/10.1017/S1471068419000152. 106

[9] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. SAFARI: smt-based abstraction for arrays with interpolants. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 679–685, 2012. 79

[10] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Booster: An acceleration-based verification framework for array programs. In *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, pages 18–23, 2014. 79

[11] Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In Antoine Miné and David Schmidt, editors, *Static Analysis*, pages 405–421, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33125-1. 82

[12] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015. 107

[13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Em-

ina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 40:1–25, 2015. 80

[14] Daphna Amit, Noam Rinetzky, Tom Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. URL `cav07`. 56, 117

[15] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pages 268–283, 2001. 79

[16] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi: 10.1007/978-3-642-22110-1_14. URL `https://doi.org/10.1007/978-3-642-22110-1_14`. 41, 42, 53, 116

[17] Amir M. Ben-Amram. Size-change termination with difference constraints. *ACM Trans. Program. Lang. Syst.*, 30(3), May 2008. ISSN 0164-0925. 113, 119

[18] M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. 42, 43, 53

[19] Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. SMT-LIB sequences and regular expressions. In *10th International Workshop on Satisfiability Modulo Theories , SMT 2012, Manchester, UK, June 30 - July 1, 2012*, pages 77–87, 2012. 63

[20] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 105–125, 2013. 79

[21] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 70–87, 2011. 79, 81, 117

[22] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. Templates and recurrences: Better together. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 688–702, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. 82

[23] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2014. 114, 119

[24] Roberto Bruttomesso, Silvio Ghilardi, and Silvio Ranise. Quantifier-free interpolation of a theory of arrays. *Logical Methods in Computer Science*, 8(2), 2012. doi: 10.2168/LMCS-8(2:4)2012. URL `https://doi.org/10.2168/LMCS-8(2:4)2012`. 79

[25] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013. ISSN 0001-0782. 73

[26] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. 56(2):82–90, 2013. 43

[27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. 17, 18, 39, 53, 56, 73, 108, 117

[28] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs by tiling. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, pages 428–449, 2017. doi: 10.1007/978-3-319-66706-5_21. URL https://doi.org/10.1007/978-3-319-66706-5_21. 80

[29] W. Checn and J. T. Duding. Program inversion: More than fun! *Science of Computer Programming*, 15(1):1 – 13, 1990. ISSN 0167-6423. 40, 115

[30] Maria Christakis and Patrice Godefroid. Proving memory safety of the ANI windows image parser using compositional exhaustive testing. URL vmcai15. 56, 117

[31] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. Sound loop superoptimization for google native client. 55, 116

[32] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. Symbolic testing of OpenCL code. . URL hvc11. 56, 117

[33] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. Symbolic crosschecking of floating-point and SIMD code. . URL eurosys11. 56, 117

[34] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Invariants for finite instances and beyond. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 61–68, 2013. URL http://ieeexplore.ieee.org/document/6679392/. 79

[35] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. 63, 81, 117

[36] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282, 1979. 9, 13

[37] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, page 84–96, New York, NY, USA, 1978. Association for Computing Machinery. ISBN 9781450373487. 82

[38] Yaniv David and Eran Yahav. Tracelet-based code search in executables. URL `pldi14`. 56, 117

[39] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. 18, 41, 42, 74, 108, 116

[40] Saumya K. Debray and Nai-Wei Lin. Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst.*, 15(5):826–875, November 1993. ISSN 0164-0925. 82

[41] Edsger W. Dijkstra. Program inversion. In *Program Construction, International Summer Schoo*, pages 54–57, London, UK, UK, 1979. Springer-Verlag. ISBN 3-540-09251-X. URL `http://dl.acm.org/citation.cfm?id=647639.733360`. 39, 115

[42] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, pages 236–254, 2000. 78

[43] Azadeh Farzan and Victor Nicolet. Modular divide-and-conquer parallelization of nested loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 610–624, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314612. URL `http://doi.acm.org/10.1145/3314221.3314612`. 80

[44] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA,*

July 15-18, 2019, Proceedings, Part I, pages 259–277, 2019. doi: 10.1007/978-3-030-25540-4_14. URL https://doi.org/10.1007/978-3-030-25540-4_14. 79

[45] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015. 81

[46] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *THEORETICAL COMPUTER SCIENCE*, 256(1):2001, 1998. 114, 119

[47] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001. doi: 10.1016/S0304-3975(00)00102-X. URL https://doi.org/10.1016/S0304-3975(00)00102-X. 77

[48] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 191–202, 2002. doi: 10.1145/503272.503291. URL http://doi.acm.org/10.1145/503272.503291. 79

[49] Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. volume 9995, pages 254–273, 11 2016. ISBN 978-3-319-48988-9. 82, 86, 109, 112, 118

[50] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993. ISBN 978-94-011-1793-7. doi: 10.1007/978-94-011-1793-7_4. URL https://doi.org/10.1007/978-94-011-1793-7_4. 10

[51] Vijay Ganesh. *Decision Procedures for Bit-vectors, Arrays and Integers*. PhD thesis, Stanford, CA, USA, 2007. AAI3281841. 18

[52] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Program-*

ming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 499–512, 2016. 40, 115

[53] Silvio Ghilardi and Silvio Ranise. MCMT: A model checker modulo theories. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, pages 22–29, 2010. 79

[54] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. Jst: An automatic test generation tool for industrial java applications with strings. 42

[55] Robert Glück and Masahiko Kawabe. A method for automatic program inversion based on lr(0) parsing. *Fundam. Inform.*, 66:367–395, 07 2005. 40, 115

[56] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. 55, 116

[57] Benny Godlin and Ofer Strichman. Regression verification. URL `dac09`. 56, 117

[58] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. 23(3):241–258, 2013. URL `stvr`. 56, 117

[59] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83, 1997. 79

[60] Radu Grigore and Hongseok Yang. Abstraction refinement guided by a learnt probabilistic model. *SIGPLAN Not.*, 51(1):485–498, January 2016. ISSN 0362-1340. doi: 10.1145/2914770.2837663. 17, 40, 115

[61] Sumit Gulwani. The reachability-bound problem. Technical Report MSR-TR-2009-146, October 2009. URL `https://www.microsoft.com/en-us/research/publication/the-reachability-bound-problem/`. 113, 118

[62] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330, 2011. 40, 115

[63] Sumit Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*, pages 9–14, New York, NY, USA, 2016. Springer-Verlag New York, Inc. ISBN 978-3-319-40228-4. 40, 115

[64] Sumit Gulwani. Programming by examples (and its applications in data wrangling). In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Verification and Synthesis of Correct and Secure Systems*. IOS Press, 2016. 80

[65] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 127–139. ACM. 11, 82, 113, 118

[66] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 375–385, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. 113, 118

[67] Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. Smt-based verification of parameterized systems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 338–348, 2016. doi: 10.1145/2950290.2950330. URL http://doi.acm.org/10.1145/2950290.2950330. 79

[68] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, pages 248–266, 2018. 75, 79

[69] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. URL esec-fse13. 56, 117

[70] Kihong Heo, Hakjoo Oh, and Hongseok Yang. Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 237–256, 2016. 40, 115

[71] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, January 1983. ISSN 0001-0782. doi: 10.1145/357980.358001. URL https://doi.org/10.1145/357980.358001. 10

[72] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735. 9

[73] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs (extended version). 03 2010. 82

[74] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012. 85, 86, 110

[75] Qinheping Hu and Loris D&#039;Antoni. Automatic program inversion using symbolic transducers. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 376–389, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062345. URL http://doi.acm.org/10.1145/3062341.3062345. 40, 115

[76] Jinru Hua and Sarfraz Khurshid. Edsketch: Execution-driven sketching for java. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 162–171. ACM, 2017. 80

[77] Oren Ish-Shalom, Shachar Itzhaky, Roman Manevich, and Noam Rinetzky. Harnessing static analysis to help learn pseudo-inverses of string manipulating procedures for automatic test generation. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 180–201, Cham, 2020. Springer International Publishing. ISBN 978-3-030-39322-9. 17

[78] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Putting the squeeze on array programs: Loop verification via inductive rank reduction. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, volume 11990 of *Lecture Notes in Computer Science*, pages 112–135. Springer, 2020. 84, 114, 119

[79] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Putting the squeeze on array programs: Loop verification via inductive rank reduction. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, volume 11990 of *Lecture Notes in Computer Science*, pages 112–135. Springer, 2020. doi: 10.1007/978-3-030-39322-9_6. URL https://doi.org/10.1007/978-3-030-39322-9_6. 57

[80] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Runtime complexity bounds using squeezers. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 320–347. Springer, 2021. doi: 10.1007/978-3-030-72019-3_12. URL https://doi.org/10.1007/978-3-030-72019-3_12. 82

[81] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 145–164. ACM, 2016. 80

[82] Pedro GarcAa Jose Oncina and Enrique Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):448–458, 1993. 9, 12, 13, 23, 36

[83] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 645–659, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 78

[84] Aditya Kanade, Rajeev Alur, Sriram Rajamani, and Ganesan Ramanlingam. Representation dependence testing using program inversion. In *Proceedings of the*

Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 277–286, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882332. URL http://doi.acm.org/10.1145/1882291.1882332. 40, 115

[85] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. Computing summaries of string loops in c for better testing and refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 874–888, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314610. URL https://doi.org/10.1145/3314221.3314610. 42

[86] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 583–602, 2015. 79

[87] Masahiko Kawabe and Robert Glück. The program inverter lrinv and its structure. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages*, pages 219–234, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. 40, 115

[88] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-338-9. doi: 10.1145/1572272.1572286. URL http://doi.acm.org/10.1145/1572272.1572286. 41, 42, 43, 53, 116

[89] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19 (7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252. 9

[90] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking

for recursive programs. *CoRR*, abs/1405.4028, 2014. URL http://arxiv.org/abs/1405.4028. 108

[91] Daniel Kroening and Michael Tautschnig. Cbmc – c bounded model checker. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54862-8. 8

[92] Shrawan Kumar, Amitabha Sanyal, R. Venkatesh, and Punit Shah. Property checking array programs using loop shrinking. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, pages 213–231, 2018. doi: 10.1007/978-3-319-89960-2_12. URL https://doi.org/10.1007/978-3-319-89960-2_12. 78

[93] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. URL pldi09. 56, 117

[94] Shuvendu K. Lahiri and Randal E. Bryant. Constructing quantified invariants via predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, pages 267–281, 2004. 79

[95] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. URL cav12. 56, 117

[96] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. 38

[97] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, page

81–92, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133367. 113, 118, 119

[98] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 401–414, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37411-4. 113, 119

[99] Kenneth L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 123–136, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37411-4. 81, 117

[100] Edison Mera, Pedro López-García, Germán Puebla, Manuel Carro, and Manuel V Hermenegildo. Combining static analysis and profiling for estimating execution times. In *International Symposium on Practical Aspects of Declarative Languages*, pages 140–154. Springer, 2007. 112, 118

[101] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *Proc. ACM Program. Lang.*, 2(POPL): 1:1–1:30, December 2017. ISSN 2475-1421. doi: 10.1145/3158089. URL http://doi.acm.org/10.1145/3158089. 40, 115

[102] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, Mar 2006. 63

[103] David Monniaux and Laure Gonnord. Cell morphing: From array programs to array-free horn clauses. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 361–382, 2016. 79

[104] David Monniaux and Laure Gonnord. Cell morphing: From array programs to array-free horn clauses. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 361–382, 2016. doi: 10.1007/978-3-662-53413-7_18. URL https://doi.org/10.1007/978-3-662-53413-7_18. 78

[105] George C. Necula. Translation validation for an optimizing compiler. URL pldi00. 56, 117

[106] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. 9, 12, 23, 38

[107] Aditya V. Nori and Sriram K. Rajamani. Program Analysis and Machine Learning: A Win-Win Deal. In *Proceedings of the 18th International Static Analysis Symposium*, volume 6887 of *Lecture Notes in Computer Science*, pages 2–3. Springer International Publishing, 2011. 40, 115

[108] Aditya V. Nori and Rahul Sharma. Termination proofs from tests. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 246–256, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491413. 17, 40, 115

[109] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 469–484, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837661. 17

[110] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 572–588, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814309. 40, 115

[111] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015. 80, 81

[112] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. URL `fse08`. 56, 117

[113] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. URL `tacas98`. 56, 117

[114] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 78

[115] Amir Pnueli, Yoav Rodeh, Ofer Strichman, and Michael Siegel. The small model property: How small can it be? 184(1):227, 2003. URL academic-ic. 56, 117

[116] Pritom Rajkhowa and Fangzhen Lin. Extending VIAP to handle array programs. In *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers*, pages 38–49, 2018. doi: 10.1007/978-3-030-03592-1_3. URL https://doi.org/10.1007/978-3-030-03592-1_3. 80

[117] David A. Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. URL cav11. 56, 117

[118] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 761–774, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837671. 17, 40, 115

[119] Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas. The four-colour theorem. *Journal of Combinatorial Theory, Series B*, 70(1):2 – 44, 1997. ISSN 0095-8956. doi: https://doi.org/10.1006/jctb.1997.1750. URL http://www.sciencedirect.com/science/article/pii/S0095895697917500. 8

[120] Brian J. Ross. Running programs backwards: The logical inversion of imperative computation. *Formal Aspects of Computing*, 9(3):331–348, May 1997. 40, 115

[121] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart and James L. Mcclelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, pages 318–362. MIT Press, Cambridge, MA, 1986. 9

[122] Hanan Samet. Compiler testing via symbolic interpretation. URL `acm-annual76`. 56, 117

[123] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 295–306, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390666. 17, 40, 115

[124] Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Mining library specifications using inductive logic programming. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 131–140, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368107. 17, 40, 115

[125] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. . 42

[126] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. . URL `issta09`. 55, 116

[127] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. 55, 116

[128] Berry Schoenmakers. Inorder traversal of a binary heap and its inversion in optimal time and space. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, pages 291–301, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. 40, 115

[129] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. 42

[130] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. URL `oopsla13`. 56, 117

[131] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, April 2012. ISSN 2150-8097. doi: 10.14778/2212351.2212356. 40, 115

[132] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reasoning*, 59(1):3–45, 2017. 113, 119

[133] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–340. ACM, 2016. 80

[134] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006. 80

[135] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. Verified three-way program merge. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):165:1–165:29, October 2018. 56, 117

[136] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. 55, 116

[137] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. From program verification to program synthesis. In *ACM Sigplan Notices*, volume 45, pages 313–326. ACM, 2010. 80

[138] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5):497–518, Oct 2013. 81, 117

[139] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):497–518, 2013. 80

[140] Ofer Strichman. Special issue: program equivalence. *Formal Methods in System Design*, 52(3):227–228, Jun 2018. 56, 117

[141] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014. 41, 116

[142] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. URL sas05. 56, 117

[143] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. 42

[144] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013. 80

[145] Yakir Vizel, Arie Gurfinkel, Sharon Shoham, and Sharad Malik. IC3 - flipping the E in ICE. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 521–538. Springer, 2017. doi: 10.1007/978-3-319-52234-0_28. URL https://doi.org/10.1007/978-3-319-52234-0_28. 10

[146] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM, 2017. 80

[147] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. Solver-based sketching of alloy models using test valuations. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 121–136. Springer, 2018. 80

[148] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *Proceedings of the ACM on Programming Languages*, 2(POPL):56:1–56:29, December 2017. ISSN 2475-1421. 56, 117

[149] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, September 1975. ISSN 0001-0782. 82, 112, 118

[150] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. S-looper: Automatic summarization for multipath string loops. 55, 116

[151] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static c analyzer. *Inf. Process. Lett.*, 102(2-3):118–123, April 2007. ISSN 0020-0190. doi: 10.1016/j.ipl.2006.11.004. 17

[152] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014. 41, 116

[153] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. 43, 53

[154] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. Z3str2: An efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design (FMSD)*, 50:249–288, June 2017. 42, 43, 53

[155] Lenore Zuck and Kenneth McMillan. *Invisible Invariants Are Neither*, pages 57–72. 09 2019. 78

[156] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In Eran Yahav, editor, *Static Analysis*, pages 280–297, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-23702-7. 113, 119