Smten and the Art of Satisfiability-Based Search

by

Richard S. Uhler

B.S., University of California, Los Angeles (2008)
S.M., Massachusetts Institute of Technology (2010)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Author .....
Department of Electrical Engineering and Computer Science
August 22, 2014

Certified by .....
Jack B. Dennis
Professor of Computer Science and Engineering Emeritus
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chairman, Committee for Graduate Students

Smten and the Art of Satisfiability-Based Search

by

Richard S. Uhler

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) have been leveraged in solving a wide variety of important and challenging combinatorial search problems, including automatic test generation, logic synthesis, model checking, program synthesis, and software verification. Though in principle SAT and SMT solvers simplify the task of developing practical solutions to these hard combinatorial search problems, in practice developing an application that effectively uses a SAT or SMT solver can be a daunting task. SAT and SMT solvers have limited support, if any, for queries described using the wide array of features developers rely on for describing their programs: procedure calls, loops, recursion, user-defined data types, recursively defined data types, and other mechanisms for abstraction, encapsulation, and modularity. As a consequence, developers cannot rely solely on the sophistication of SAT and SMT solvers to efficiently solve their queries; they must also optimize their own orchestration and construction of queries.

This thesis presents Smten, a search primitive based on SAT and SMT that supports all the features of a modern, general purpose functional programming language. Smten exposes a simple yet powerful search interface for the Haskell programming language that accepts user-level constraints directly. We address the challenges of supporting general purpose, Turing-complete computation in search descriptions, and provide an approach for efficiently evaluating Smten search using SAT and SMT solvers. We show that applications developed using Smten require significantly fewer lines of code and less developer effort for results comparable to standard SMT-based tools.

Thesis Supervisor: Jack B. Dennis
Title: Professor of Computer Science and Engineering Emeritus

Acknowledgments

I could not have completed this thesis without the help of my advisors, colleagues, family, and friends. It has been a particular challenge of this thesis to tell a coherent and digestible story of Smten. At times it seemed I would never finish, as I inevitably circled back to previous attempts at explaining Smten, but thanks to substantial feedback and the emotional support of others, the story of Smten steadily improved and has been put to text in this document.

I am indebted to my thesis advisor, Jack Dennis, who provided encouragement and support as my research topic meandered before ultimately settling somewhat afield from where it began. Jack's advice and optimism were uplifting when I was in low spirits, and he reviewed many repeated drafts of my writing in great detail, often on short notice.

Smten would not have seen the light of day without Nirav Dave, who convinced me that making SMT solvers easier to use was a worthwhile thesis topic. Throughout the life of Smten, Nirav has served in the roles of salesman, advocate, advisor, early user, coauthor, and manager. His guiding vision and abundant feedback have been critical in making Smten the success it has been.

I have been fortunate to have Arvind and Armando Solar-Lezama on my thesis committee alongside Jack and Nirav. Three semesters as Arvind's teaching assistant for 6.375 on Complex Digital Systems helped develop my presentation skills and instilled in me key concepts from Bluespec that have carried over to Smten. Arvind offered his thoughts on many of my presentations of Smten over the years. Armando provided valuable suggestions for how to evaluate Smten and how to clarify the presentation of Smten.

I would like to thank Peter Neumann and SRI International for graciously supporting me for two summers as I worked on my thesis.

Additionally I would like to acknowledge members of the Computation Structures Group: Murali Vijayaraghavan for valuable technical discussions, Myron King as an early user of Smten, Asif Khan, Abhinav Agarwal, Andy Wright, and Thomas

Bourgeat for feedback on my papers and presentations, and everyone for their companionship and stimulating lunch conversations.

Finally, I would like to express my gratitude for my family, Chuck and Linda Nielsen, Stephen Uhler, and Jessie Uhler, who contributed moral support and advice on how to finish my thesis. I would especially like to thank Elizabeth Forsyth for providing emotional support, helping with all aspects of the world outside of my thesis, and editing my publications and dissertation.

The research in this thesis was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 and supported by National Science Foundation under Grant No. CCF-1217498. The views, opinions, and/or findings contained in this report are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the Department of Defense, or the National Science Foundation.

Contents

List of Figures

Chapter 1

Introduction

I implemented my first Sudoku solver shortly after becoming a graduate student at the Massachusetts Institute of Technology. I had recently been introduced to functional programming and wanted a better sense of how it compares to C++ and Java, the imperative languages I was most familiar with. My plan was to implement the same program in both an imperative programming language and a functional programming language. Sudoku was an attractive application to use because the problem statement is simple; it exercises core language features in a self-contained, but reasonably complex manner; and I had been playing the game in my free time.

Sudoku is a logic puzzle commonly found in newspapers, alongside the crossword puzzles, anagrams, and other games. You are given a partially completed grid of digits with nine rows and nine columns, partitioned into nine separate square areas called boxes. Your task is to place a digit in each of the blank cells of the grid so that each row, column, and box contains all unique digits from 1 through 9.

I implemented my Sudoku solver in C++ using a brute-force approach: try all possible digits for each blank cell in turn. Much to my surprise, my Sudoku solver worked and was *fast*, solving a Sudoku puzzle in less than a second. After that, playing Sudoku as a past time lost its appeal to me. What is interesting about a puzzle that a computer can solve so easily? A Sudoku solver is not a terribly useful tool to have in life, and as I found out, it is not difficult to implement a reasonably fast solver. Regardless, Sudoku served its purpose in my explorations of functional

and imperative programming.

As it turns out, there are many applications involving combinatorial search, like Sudoku, that are both useful and much harder for a computer to solve than Sudoku:

Automatic Test Generation is used in software development. Instead of searching for digits to place in blank cells, an automatic test generator searches for inputs to a program that exercise a particular block of code.

Logic Synthesis is used for the optimization of digital circuits. It involves searching for better configurations of logic gates with the same behavior as an initial configuration of gates.

String Constraint Solving is used to synthesize character strings satisfying a given set of constraints. String solvers can be used to suggest passwords meeting obscure hardness constraints or to generate SQL queries that test for SQL injection attacks, for instance.

Model Checking is used for formal verification of software and hardware systems. The model checking problem can be posed as a search for system traces that violate properties specified by the user, often in the form of temporal logic.

Program Synthesis is when a computer is responsible for writing or finishing an implementation of a program to meet a designer's specifications. It also falls into the category of combinatorial search applications.

For applications such as these, the brute-force approach I took to implement my Sudoku solver is not nearly good enough. These NP-hard problems are theoretically intractable to solve by computer as the problem sizes grow, and unlike Sudoku, tools that solve these problems must handle large problem sizes to be useful in practice. Fortunately, there is a different general approach to solving combinatorial search problems that can be used to tackle practically sized problems. I call it the *satisfiability-based* approach to combinatorial search.

The idea behind satisfiability-based search is to leverage a primitive search operation that solves queries of the form:

$$\exists x.\, \phi(x)$$

The primitive search operation searches for a value of the variable x that satisfies constraints described by the formula $\phi(x)$. In the satisfiability-based approach to search, developers decompose their high-level search problem into primitive search queries. This is more effective than a brute-force approach because it leverages a highly-optimized search procedure built by experts to evaluate the primitive search queries.

The past decade or so has seen an enormous growth in the satisfiability-based search approach, thanks to the availability of high performance Satisfiability (SAT) solvers and variations that serve as the optimized procedure for evaluating search queries. The approach has been employed for automatic test generation [8], automatic theorem proving [37], logic synthesis [42], model checking [5, 41, 48], program synthesis [50], quantum logic synthesis [27], string constraint solving [31], and software verification [60].

Unfortunately, current limitations in the functionality and interfaces of the solvers used for the primitive search query, $\exists x.\, \phi(x)$, mean substantial development and engineering effort is often required before tools implemented with the satisfiability-based approach work well enough to be useful in practice. There are two varieties of solver used to implement primitive search: Satisfiability (SAT) solvers and Satisfiability Modulo Theories (SMT) solvers. SAT solvers support queries for $\exists x.\, \phi(x)$ where x is a finite collection of boolean variables and $\phi(x)$ describes constraints solely in terms of logical AND, OR, and NOT operations. SMT solvers support some other kinds of variables and constraints, depending on the background theories of the solver, such as integer variables and constraints involving arithmetic operations and comparisons, bit-vectors variables with constraints involving bitwise operations, or fixed-length array variables with constraints involving read and write operations. SAT and SMT

solvers fail to support constraints described using the wide array of features programmers rely on for general purpose programming: procedure calls, loops, recursion, user-defined data types, recursively defined data types, and other mechanisms for abstraction, encapsulation, and modularity.

Developers rely on the general purpose programming features of a separate host language to dynamically construct the more limited form of search query supported by SAT and SMT solvers. This indirection causes significant headache for the development of practical satisfiability-based tools and requires the developer to take special care in optimizing the construction of queries, abating the benefits of relying on the sophistication of SAT and SMT solvers to efficiently solve the queries.

Can we do away with the current limitations of solvers and provide an implementation of the search primitive with support for *all* the features of a modern, general purpose programming language? If so, how much easier would it be to apply the satisfiability-based approach to develop practically useful tools for combinatorial search applications?

These questions are the focus of this thesis. To investigate these questions, we have designed a new interface and implementation of a primitive search operation called *Smten* search. Smten search supports all the features of a modern, general purpose, and, in particular, *functional* programming language.

The implementation of Smten search is built on top of existing SAT and SMT solvers. Much of this thesis addresses the following important challenges required for Smten search:

- Design of a simple search interface capable of expressing user-level constraints that also supports efficient reduction to SAT and SMT.

- Integration of standard language features with SAT and SMT. Most notably this includes support for Turing-complete computation in search constraints, but also input/output operations, primitive type conversions, and other operations that are not directly supported by SAT or SMT solvers.

- Development of an implementation of Smten search with good practical perfor-

mance.

To evaluate the effect of Smten search on development effort and performance, we used Smten search to reimplement three tools for different combinatorial search applications: string constraint solving, model checking, and program synthesis. For each tool there exists a state-of-the-art satisfiability-based implementation using SAT and SMT search directly. We compared our implementations of the tools to the existing implementations. Experiments show that Smten search can reduce developer effort by orders of magnitude over direct SAT and SMT search while achieving comparable overall performance.

1.1 Thesis Contributions

The specific contributions of this thesis are:

- We present a programming abstraction for Smten search that is amenable to automatic reduction to the problem of satisfiability (Chapter 3).

- We distinguish among the different ways Turing-complete computation can affect search-space descriptions, and discuss what behaviors of search in these search-space descriptions are suitable for modularity, expressiveness, and performance (Chapter 3).

- We provide operational semantics for Smten search to clarify the behavior of search for search-space descriptions involving Turing-complete computation. The semantics are given solely in terms of the programming abstraction exposed to the user, and do not rely upon or assume an underlying behavior from a SAT or SMT solver (Chapter 3). The semantics have been formalized and mechanically checked using the Coq proof assistant.

- We give a syntax directed approach that shows what needs to be added to standard functional programming languages to support efficient execution of Smten search by SAT and SMT (Chapter 4).

- We distinguish between reachable and unreachable nonterminating computation during search evaluation, and suggest that both cases should be handled with the same mechanism (Chapter 4).

- We describe an abstraction-refinement procedure for efficiently supporting both reachable and unreachable nonterminating computation during search evaluation (Chapter 4).

- We have built a fully featured compiler for Smten based on the Glasgow Haskell Compiler (Chapter 5).

- We have developed three substantial satisfiability-based search applications using the Smten language and compiler: a model checker, a string constraint solver, and a program synthesis tool (Chapter 6).

- We compare the Smten-based implementation with state-of-the art hand-coded implementations of the model checker, string constraint solver, and program synthesis tool (Chapter 6).

1.2 Thesis Organization

The remainder of this document is organized as follows:

Chapter 2 Discusses the background of the satisfiability-based approach to combinatorial search. It introduces the example of a string constraint solver as a combinatorial search problem, illustrates how the satisfiability-based approach can be used to implement a string constraint solver, and discusses the practical limitations of using SAT and SMT solvers for the underlying primitive search implementation.

Chapter 3 Describes the user interface of the Smten search primitive. It begins with a review of the Haskell programming language, then introduces the Smten search primitives for describing search spaces and executing search queries. The

chapter includes a sequence of examples and discussion to illustrate the different ways Turing-complete computation can be used in describing search spaces and motivate the desired behavior of search in the presence of Turing-complete search descriptions. Finally we present an operational semantics for the Smten search interface in terms of a kernel language for Smten.

Chapter 4 Describes an approach for implementing the Smten search primitive based on SAT and SMT solvers. It walks through an example illustrating the high-level approach to search in Smten before presenting formal evaluation rules for the search implementation. Implementation-level issues with nontermination are discussed and a mechanism is presented for handling Turing-complete computation in search-space descriptions. The chapter concludes with a description of additional optimizations to the search approach that are important in practice.

Chapter 5 Describes a compiler for the Smten language based on the Glasgow Haskell Compiler.

Chapter 6 Provides an evaluation of Smten search. It presents our experience developing a string constraint solver, model checker, and program synthesis tool using Smten, and comparison of these applications with corresponding state-of-the-art hand coded SAT and SMT based implementations of the same tools.

Chapter 7 Summarizes the results and status of Smten search, suggests avenues of continuing research, and makes concluding remarks.

Chapter 2

Satisfiability-Based Search

In this chapter we provide background on the satisfiability-based approach to solving combinatorial search problems. Our goal is to understand the advantages of the satisfiability-based approach for performance and development effort in principle and the challenges that exist for satisfiability-based search in practice.

Figure 2-1 shows a stylized plot of the performance and development effort for different approaches to search on practically sized problems. The performance axis represents the end-to-end time to solve a combinatorial search problem. An approach to search that requires less time to solve a problem has higher performance. In the figure, the performance axis is marked as log scale to reflect that changing the approach to search can improve performance by orders of magnitude. Often these orders of magnitude can be the difference between a tool that fails entirely and a tool that works well on practically sized problems.

The horizontal axis, labeled "Modularity/Simplicity/Flexibility", is intended to represent the developer effort required to implement a search application. An approach to search with more modularity, simplicity, and flexibility leads to applications that require less developer effort to implement. Modularity, simplicity, and flexibility are difficult to quantify, but intuitively they lower the costs to develop, understand, debug, and maintain an application, and they lower the costs to add new features or explore alternate design choices. We will present many examples to give a qualitative sense of what modularity, simplicity, and flexibility mean for the implementation of

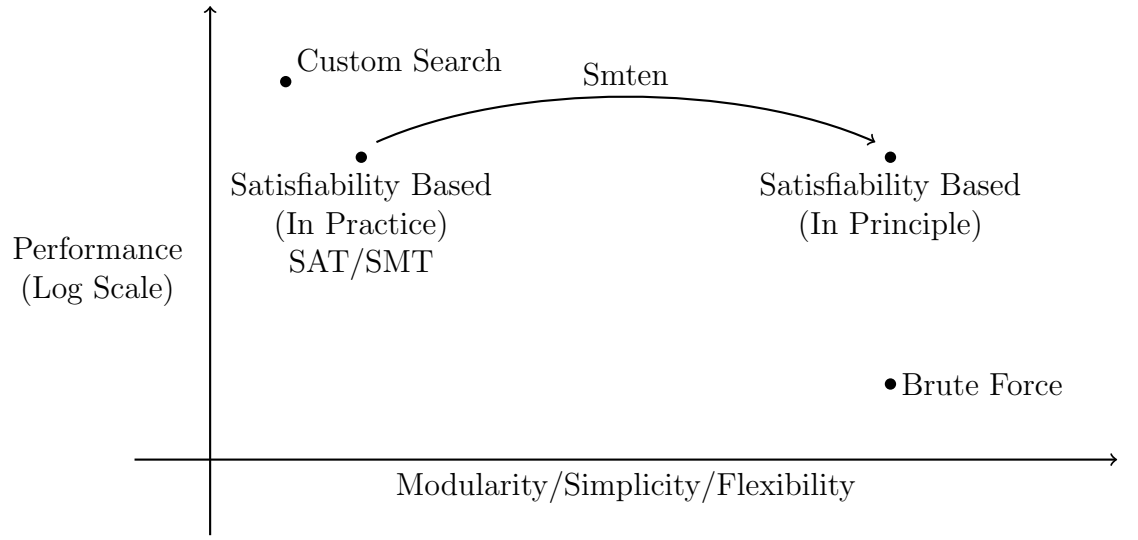

Figure 2-1: Performance and development effort for different approaches to search.

combinatorial search problems.

The simplest approach to search is a brute-force approach, which enumerates all possible solutions in order to find a result. The brute-force approach to search is modular and flexible but has terrible performance and is practically useless for interesting problems.

A custom approach to search leverages specialized algorithms for the specific problem being solved to achieve the best possible performance. However, developing and implementing good algorithms for a specific problem can require decades of research effort, and the algorithms tend to work under specific assumptions about the problem that make it difficult to support new features.

In principle the satisfiability-based approach to search provides the modularity, simplicity, and flexibility of brute-force search with much of the benefits of a custom search algorithm. This is because it leverages a specialized solver for satisfiability that supports general constraints. In practice, however, the satisfiability-based approach can be difficult to use because of limitations in how constraints are described to SAT and SMT solvers. Developers rely on the general purpose programming features of a separate host language to dynamically construct the more limited form of search query supported by SAT and SMT solvers. This indirection causes significant headache for

the development of practical satisfiability-based tools and requires the developer to take special care in optimizing the construction of queries, abating the benefits of relying on the sophistication of SAT and SMT solvers to efficiently solve the queries.

The goal of this thesis is to realize the full potential of satisfiability-based search using Smten, which supports all features of a general purpose programming language for describing constraints.

Section 2.1 introduces a specific problem of string constraint solving that we will use to discuss satisfiability-based search in more detail. We will use this string constraint solving as a running example throughout the thesis. After first looking at brute-force (Section 2.2) and custom (Section 2.3) approaches to implementing the string constraints solver, we will show how the satisfiability-based approach can be applied to solving the problem, and the benefits it provides in principle (Section 2.4). To give a sense of the broader uses of the satisfiability-based approach to search, we will show how the approach has been applied to model checking and program synthesis as well.

We will review SAT and SMT solvers (Section 2.5), which are at the core of the satisfiability-based approach to search, before discussing in some depth the practical challenges of using SAT and SMT solvers for satisfiability-based search (Section 2.6). This will put us in a position to present the idea of Smten search, an enhanced form of satisfiability-based search without the limitations of SAT and SMT search (Section 2.7)

We conclude the chapter with a brief survey of related work (Section 2.8).

2.1 A String Constraint Solver

To illustrate issues with performance and developer effort in combinatorial search applications, consider the example of string constraint solving.

The goal of a string constraint solver is to synthesize a character string that satisfies a set of constraints. Possible uses of a string constraint solver include:

Email Address Suggestion For example, synthesize a string that is less than 16

characters, contains the substring `"ruhler"`, and forms a valid MIT email address. Possible solutions include `"ruhler00@mit.edu"`, `"ruhler@mit.edu"`, and `"bzruhler@mit.edu"`.

Password Suggestion For example, synthesize a string that has exactly 8 characters, contains at least one lowercase character, uppercase character, number, and symbol, and does not belong to a list of commonly used passwords.

Test Case Generation For example, synthesize a string of less than 256 characters that is a valid SQL query and contains the substring `"OR '1'='1'"`, commonly used in SQL injection attacks.

The specific string constraint solver we will look at synthesizes a string of characters satisfying the following three kinds of constraints:

Length The length constraint `length l to h`, where l and h are nonnegative integers, specifies that the synthesized string should have at least l characters and at most h characters. For example, the length constraint `length 2 to 4` is satisfied by strings `"ab"`, `"foo"`, and `"blah"`, but not by strings `"x"` or `"sludge"`.

Substring The substring constraint `contains s`, with a literal string s, specifies that the synthesized string should contain s as a substring. For example, the substring constraint `contains "foo"` is satisfied by the strings `"foobar"`, `"seafood"`, and `"foo"` itself, but not by the strings `"abcdef"`, `"floor"`, or `"spoof"`.

Regular Expression The regular expression constraint `matches r` specifies that the synthesized string should belong to the language of the regular expression r. For example, the regular expression constraint `matches (ab)*(cd)*` is satisfied by strings with zero or more repetitions of the character sequence `"ab"` followed by zero or more repetitions of the character sequence `"cd"`. This includes the strings `"ababcdcd"`, `"abababab"`, and `"cd"`, but not the strings `"abcdef"`, `"aba"`, or `"cdab"`. We will define the syntax and interpretation of regular ex-

pressions more precisely when we present a brute-force implementation of the string constraint solver in Section 2.2.

The input to the string constraint solver is a *string constraint specification*, which describes a set of zero or more of each kind of constraint, all of which must be satisfied by the synthesized string. The following example shows a string constraint specification for the email address suggestion example, where semicolons are used to separate the individual constraints:

```
length 0 to 16; contains "ruhler"; matches [a-z]*@mit.edu;
```

In a more complicated synthesis problem, there can be multiple instances of each kind of constraint. For example, to synthesize a string of any length less than 16 that contains the substrings `"foo"`, `"arf"`, and `"bar"`, the following string constraint specification could be used:

```
length 0 to 16; contains "foo"; contains "arf"; contains "bar";
```

Strings satisfying this constraint include `"fooarfbar"`, `"foodtarfibare"`, and the string `"barfoo"`.

In general, multiple occurrences of the `length` constraint can be merged into a single `length` constraint by taking the largest lower bound and smallest upper bound of the `length` constraints. For example, `length 4 to 8; length 6 to 12` is equivalent to `length 6 to 8`. For this reason, we will assume there is no more than one `length` constraint in a string constraint specification.

Though regular expression constraints cannot be merged directly like the length constraints can, we also assume there is no more than one regular expression constraint in the string constraint specification. This simplifies the problem and opens up an alternate interpretation for the regular expression constraint as a structured template for the string to synthesize. For example, the following string constraint specification will yield a string with the structure of a web address containing the

substring `"ru"`:

```
matches http://www.[a-z][a-z][a-z].(com|edu|net); contains "ru"
```

Note that it is possible to over-constrain the synthesis problem so that there are no solutions. For example:

```
length 3 to 3; contains "foo"; contains "bar";
```

For an over-constrained string such as this, the string constraint solver should report that there is no solution.

The string constraint solving problem we have defined is a combinatorial search problem; as the string length increases, the number of candidate strings grows exponentially. String constraint solving serves as a good example of combinatorial search because it is simple enough to understand and present the implementation while being complex enough to illustrate all the important challenges involved in implementing a combinatorial search application.

We mentioned earlier that modularity and flexibility are important considerations for the approach used to implement search. The string constraint solver, which could support a variety of different kinds of constraints, illustrates why this is the case. For instance, the password suggestion example is not easily expressed in terms of substring and regular expression constraints; ideally it would be a simple task to adapt the string solver implementation to support a password suggestion kind of constraint.

2.2 Brute-Force Search

The simplest approach to implementing the string constraint solver is to use a brute-force approach: enumerate all possible strings, filtering out those strings that fail to satisfy the constraints one-by-one until a satisfactory solution is found.

The brute-force approach is simple and modular because the search approach is

decoupled from the description of the constraints, the kinds of constraints can each be considered in isolation, and highly-tuned libraries can be reused for evaluating the constraints. The organization of the brute-force search can also be adjusted to improve performance without affecting the implementation of the constraints. A benefit of reviewing the brute-force approach is that it shows what information, at minimum, is required to completely specify a search problem.

Figure 2-2 shows a brute-force algorithm for the string constraint solver, illustrating that the approach to search is decoupled from the specific constraints supported by the solver.

The input to the procedure STRSOLVE (Line #1) is l, the lower bound on the string length, h, the upper bound on the string length, the single regular expression constraint r, and the set X of substrings that the synthesized string must contain. The algorithm uses the procedure STRS (Line #15) to enumerate each possible string of a given length, then uses the procedure WELLCONSTRAINED (Line #29) to test whether a candidate string s satisfies the regular expression constraint r and all of the substring constraints X. If there is no solution, the STRSOLVE procedure indicates this by returning NoSuchString (Line #12).

The STRS procedure recursively computes the set of strings of length i. The base case, when i is 0, is a singleton set with the empty string, denoted as ϵ (Line #17) Otherwise the set S' is constructed by including all strings formed by the concatenation of a, a single character, with a string b of length $i-1$. The notation ab is used for concatenation of the strings a and b (Line #23). Notice that we do not need to know how WELLCONSTRAINED is implemented to understand the procedure in Figure 2-2.

The implementation of WELLCONSTRAINED differs depending on the kinds of constraints supported by our string constraint solver. A benefit of the brute-force approach to search is that we can consider each kind of constraint separately when implementing WELLCONSTRAINED. The following sections show what is required to describe the substring and regular expression constraints specific to our version of the string constraint solver.

```
1: procedure STRSOLVE(l, h, r, X)
2:     i ← l
3:     while i ≤ h do
4:         S ← STRS(i)
5:         for s ∈ S do
6:             if WELLCONSTRAINED(r, X, s) then
7:                 return s
8:             end if
9:         end for
10:        i ← i + 1
11:    end while
12:    return NoSuchString
13: end procedure
14:
15: procedure STRS(i)
16:    if i = 0 then
17:        return {ϵ}
18:    end if
19:    S ← STRS(i − 1)
20:    S′ ← ∅
21:    for a ∈ Char do
22:        for b ∈ S do
23:            S′ ← S′ ∪ {ab}
24:        end for
25:    end for
26:    return S′
27: end procedure
28:
29: procedure WELLCONSTRAINED(r, X, s)
30:    ...
31: end procedure
```

Figure 2-2: Brute-force implementation of the string constraint solver.

```
1: procedure CONTAINS(x, s)
2:     if x = ε then
3:         return True
4:     end if
5:     i ← 1
6:     while i ≤ |s| do
7:         if PREFIX(x, s_{i,...,|s|}) then
8:             return True
9:         end if
10:        i ← i + 1
11:    end while
12:    return False
13: end procedure
14:
15: procedure PREFIX(x, s)
16:    if |x| > |s| then
17:        return False
18:    end if
19:    i ← 1
20:    while i ≤ |x| do
21:        if x_i ≠ s_i then
22:            return False
23:        end if
24:        i ← i + 1
25:    end while
26:    return True
27: end procedure
```

Figure 2-3: Procedure for testing a substring constraint.

2.2.1 Substring Constraint

Figure 2-3 shows a procedure to verify a candidate string satisfies a substring constraint. The procedure CONTAINS (Line #1) verifies the string s contains x as a substring by iterating through the positions of s searching for x. The PREFIX procedure (Line #15) tests whether the string x is a prefix of s. The notation $|s|$ denotes the length of the string s, s_i denotes the ith character of the string s, and $s_{i,\ldots,|s|}$ is the substring of s starting at the ith character and including all remaining characters of s.

$$
\begin{array}{rrl}
 & c & \in \quad Char \\
\text{RegEx} & r & ::= \quad \emptyset \;\mid\; \epsilon \;\mid\; c \;\mid\; r_1\, r_2 \;\mid\; r_1 \mid r_2 \;\mid\; r*
\end{array}
$$

Figure 2-4: Syntax of regular expressions.

$$
\begin{array}{rcl}
\mathcal{L}(\emptyset) & = & \{\} \\
\mathcal{L}(\epsilon) & = & \{\epsilon\} \\
\mathcal{L}(c) & = & \{c\} \\
\mathcal{L}(r_1\ r_2) & = & \{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1 \mid r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r*) & = & \mathcal{L}(\epsilon \mid r\ r*)
\end{array}
$$

Figure 2-5: Semantics of regular expressions.

2.2.2 Regular Expression Constraint

The regular expression constraint is more complex to describe, because it depends on the specific syntax and interpretation for regular expressions supported by the string constraint solver.

It is not critical for the reader to understand the details of regular expressions presented here. The developer of a string constraint solver, however, *is* responsible for understanding and implementing these details as part of the tool.

Figure 2-4 shows a basic regular expression syntax. The language of strings described by a regular expression r is denoted $\mathcal{L}(r)$. Figure 2-5 shows the language of strings described by each kind of regular expression. The regular expression $\emptyset$ represents the empty language; ϵ represents a single-element language containing the empty string; c represents the single-element language containing the string with c as the single character; $r_1\ r_2$ represents the language where each string is the concatenation of a string from the language of r_1 followed by a string from the language that r_2 describes; $r_1 \mid r_2$ is the language of strings in either the language of r_1 or the language that r_2 describes; and $r*$ is the language of strings that are zero or more repeated concatenations of strings from the language of r.

In practice it is convenient to define additional syntax for regular expressions that can be de-sugared into the core regular expression syntax. For example, `[`c_1`-`c_2`]` is

shorthand for any character ranging between characters c_1 and c_2; r+ means one or more repeated concatenations of strings from the language of r; and r? means either a string from the language r or the empty string.

Figure 2-6 shows an example of a recursive procedure that could be used to test whether a candidate string s matches a regular expression r. It performs a case analysis of the regular expression r. Most of the cases are straight forward. The cases for r_1 r_2 and $r*$ iterate over each partition of the string s into two disjoint substrings. This procedure is straightforward but is not very efficient. There are many advanced techniques that can be applied to improve the efficiency of the match procedure. Typically a programmer would not develop their own regular expression match procedure; instead they would reuse a highly tuned library for regular expression match. The brute-force approach to search supports reuse of library functions in this fashion.

2.2.3 Composing Constraints

The implementation of WELLCONSTRAINED in Figure 2-7 demonstrates how simple it is to compose the substring and regular expression constraints together to verify a candidate string satisfies all of the constraints.

The implementation can easily be adapted to support new kinds of constraints because it is simple to compose the constraints. Here are two examples of adaptations that could be implemented:

- It would be trivial to modify the string constraint solver implementation to support multiple regular expression constraints by changing the WELLCONSTRAINED procedure to iterate through each regular expression constraint instead of checking against a single regular expression.

- It would be easy to add a new kind of constraint that counts the number of symbols, numbers, uppercase, and lowercase characters in the candidate string to support password suggestion. A new procedure to test the password suggestion constraint must be implemented, and the WELLCONSTRAINED procedure should be modified to iterate through each of the password suggestion constraints in

```
procedure MATCH(r, s)
    if r = ∅ then
        return False
    else if r = ϵ then
        return (s = ϵ)
    else if r = c then
        return (s = c)
    else if r = r_1 r_2 then
        i ← 0
        while i ≤ |s| do
            if MATCH(r_1, s_{1,...,i}) and MATCH(r_2, s_{i+1,...,|s|}) then
                return True
            end if
            i ← i + 1
        end while
        return False
    else if r = r_1 | r_2 then
        return MATCH(r_1, s) or MATCH(r_2, s)
    else if r = r_1* then
        if s = ϵ then
            return True
        end if
        i ← 1
        while i ≤ |s| do
            if MATCH(r_1, s_{1,...,i}) and MATCH(r_1*, s_{i+1,...,|s|}) then
                return True
            end if
            i ← i + 1
        end while
        return False
    end if
end procedure
```

Figure 2-6: Procedure for testing a regular expression constraint.

```
procedure WELLCONSTRAINED(r, X, s)
    if not MATCH(r, s) then
        return False
    end if
    for x ∈ X do
        if not CONTAINS(x, s) then
            return False
        end if
    end for
    return True
end procedure
```

Figure 2-7: Procedure composing the different kinds of string constraints.

addition to the constraints it already checks.

In both of these examples, the procedures CONTAINS and MATCH are reused without modification.

2.2.4 Reorganizing the Brute-Force Search

Finally, we will show that the organization of the brute-force search can be adjusted to improve performance without affecting the implementation of constraints. The magnitude of performance improvements from reorganizing search depends on the expected use cases of the string constraint solver. The brute-force approach to search is easy to adapt to different usage scenarios by adjusting the organization of search.

For example, the STRSOLVE procedure in Figure 2-2 enumerates all strings of length between l and h from the length constraint, then tests whether the strings satisfy the regular expression and substring constraints. A different way to organize the brute-force search is to enumerate all strings from the language of the regular expression constraint, then test whether they satisfy the length and substring constraints. This organization could be much more efficient if, for example, the user is searching for a long and highly structured string.

Consider the following string constraint specification, which could be used to synthesize a web address for one of the top level domains `.edu`, `.com`, or `.net` that contains the substring `"ru"` and is 18 characters long:

```
length 18 to 18;
matches http://www.[a-z][a-z][a-z].(edu|com|net);
contains "ru";
```

Our first brute-force implementation will search over the space of all 18-character strings. With an alphabet of 26 characters, there are 26^{18}, or roughly 10^{25} strings belonging to that search space. However, there are only $3 * 26^3 = 52728$ strings in the language of the regular expression.

What organization of the search to use depends on how the designer expects his tool to be used. If the tool is used mostly for short strings with little structure, it is better to search over all the strings of a given length. If the tool is used mostly for long, highly structured strings, then it more efficient to search over all the strings belonging to the language of the regular expression.

Figure 2-8 shows the modified version of the string constraint solver that enumerates strings by regular expression instead of length. It performs a case analysis on the regular expression r, constructing the appropriate set of strings according to the regular expression semantics from Figure 2-5. The WELLCONSTRAINED function must also be modified to check for the length constraint instead of the regular expression constraint.

Aside from having to reimplement the STRS procedure, this change was simple to make. In particular, the procedure for testing the substring constraint was not affected by the change.

In summary, the brute-force approach to search is simple to implement and provides a high degree of modularity and encapsulation. The procedures for testing different kinds of constraints are independent, both from the overall search procedure and from each other. As a consequence, it is relatively easy to add support for new constraints, change the search organization, and reuse highly-tuned library procedures.

Unfortunately, the brute-force approach does not scale, because in the worst case it must enumerate and test an exponentially large number of candidate strings.

```
procedure STRSOLVE(l, h, r, X)
    S ← STRS(r)
    for s ∈ S do
        if WELLCONSTRAINED(l, h, X, s) then
            return s
        end if
    end for
    return NoSuchString
end procedure

procedure STRS(r)
    if r = ∅ then
        return ∅
    else if r = ϵ then
        return {ϵ}
    else if r = c then
        return {c}
    else if r = r1 r2 then
        S ← ∅
        for s1 ∈ STRS(r1) do
            for s2 ∈ STRS(r2) do
                S ← S ∪ {s1 s2}
            end for
        end for
        return S
    else if r = r1 | r2 then
        return STRS(r1) ∪ STRS(r2)
    else if r = r1* then
        S ← {ϵ}
        for s1 ∈ STRS(r1) do
            for s2 ∈ STRS(r) do
                S ← S ∪ {s1 s2}
            end for
        end for
        return S
    end if
end procedure
```

Figure 2-8: Alternate organization of the brute-force string constraint solver that enumerates the space of strings by regular expression.

2.3 Custom Search

A more efficient approach to implementing a string constraint solver is to design a search algorithm specific to the kinds of constraints supported by the solver. For example, assume for the time being we that are interested in implementing a string constraint solver with support for only the length and substring constraints.

It is simple to construct a string s that contains a set of substrings X by concatenating each of the substrings. For example, given substrings `"foo"`, `"arf"`, and `"bar"`, we would construct the string `"fooarfbar"`. This string contains each of the required substrings by construction. The length of the constructed string using this approach will be the sum of the lengths of each substring. If this length satisfies the upper bound on the string length specified by the user, the constructed string is a solution to the string constraint specification; if the constructed string is too short, it can be extended without violating the constraints by appending arbitrary characters to the string.

This custom search approach is much more efficient than the general brute-force approach of enumerating and testing all possible strings. However, the approach described is incomplete if the length of the string formed by construction exceeds the upper bound in the string constraint specification.

Instead of concatenating the substrings directly, a complete approach could try to construct the smallest string containing all of the substrings. Now if the constructed string is too large, then there is no solution to the string constraint specification. If the constructed string is too small, it can be extended with arbitrary characters without violating constraints. It turns out the problem of constructing the smallest string containing all of the substrings is a well known problem, called the shortest common superstring problem [21]. The shortest common superstring problem is NP-complete, meaning there are no known polynomial time algorithms to solve it. Approximate algorithms for the shortest common superstring [6, 55] could be used to implement a string constraint solver that works well on most inputs, but in the worst case, the string constraint solver may need to find an exact solution to the problem.

N	Brute Force	Satisfiability Based	Custom Search
5	<1s	<1s	<1s
10	145s	<1s	<1s
50	NA	<1s	<1s
100	NA	<1s	<1s
500	NA	20s	<1s
1000	NA	NA	<1s
5000	NA	NA	8s
10000	NA	NA	32s

Figure 2-9: Runtime of n-queens solver with different search approaches.

A custom domain approach to search, such as using algorithms for shortest common superstring, is likely to produce a much better algorithm for string constraint solving than brute force. Because there is freedom in the choice of synthesized string, approximate algorithms can be used to provide a string solver that performs well for many different string constraint specifications. The problem with a custom domain approach to search is the search procedure and constraints are tightly coupled, and the approach is tied to the specific constraints the solver supports. It would be difficult to combine approximation algorithms for shortest common superstring with finite automata approaches for regular expression constraints into a single, coherent, and effective algorithm for search involving both substring and regular expression constraints. How easy would it be, then, to add support for the password suggestion constraints to the string constraint solver?

2.4 Satisfiability-Based Search

The idea of satisfiability-based search is to provide a primitive search operation that solves queries of the form:

$$\exists x.\, \phi(x)$$

The primitive search operation searches for a value of the variable x that satisfies constraints described by the formula $\phi(x)$.

A combinatorial search problem can be implemented by decomposing it into prim-

itive search queries. The satisfiability-based approach to search combines the conceptual simplicity and modularity of the brute-force approach with a specialized, high-performance search procedure developed by experts.

For a rough sense of how effective satisfiability-based search can be, Figure 2-9 compares how well the brute force, satisfiability based, and custom approaches to search scale for the n-queens puzzle, which is to place n queens on an $n \times n$ chess board such that no queen can attack another. The brute-force approach to search can barely scale to $n = 10$, the satisfiability-based approach, which uses a single primitive search query to answer the problem, scales to around $n = 500$, while a custom search approach for n-queens using a probabilistic local search algorithm based on a gradient-based heuristic [51] scales even better.

The satisfiability-based approach is more efficient than the brute-force approach because the implementation of the search primitive includes advanced heuristics and techniques to avoid enumerating all candidate solutions, and it exploits sharing in the search computation as much as possible.

For more complicated combinatorial search problems, the custom approach to search is less feasible to implement due to high research and development costs. The satisfiability-based approach can be applied to more complicated problems using multiple primitive search queries to solve the problem. To get a sense of the variety of ways the satisfiability-based approach can be applied to more complicated combinatorial search problems, we will consider examples from string constraint solving, model checking, and program synthesis.

2.4.1 String Constraint Solving

A natural way to implement a string constraint solver using the satisfiability-based approach to search is to issue a separate primitive search query for each possible length of the desired string. The primitive search query is well suited to answer questions about fixed-length strings, and the order in which to search for different length strings can be guided by the developer.

For example, a reasonable order would be to search first for the smallest length

```
1: procedure STRSOLVE(l, h, r, X)
2:     i ← l
3:     while i ≤ h do
4:         if Sat(∃s.WELLCONSTRAINED(r, X, s_{1,...,i})) then
5:             return Witness(s)
6:         end if
7:         i ← i + 1
8:     end while
9:     return NoSuchString
10: end procedure
```

Figure 2-10: Satisfiability-based string constraint solver.

string, then for longer and longer strings until a solution is found. The primitive queries, of the form $\exists x.\, \phi(x)$, will have x be a variable for a string of fixed length, and ϕ will determine if the string x satisfies the user-level string constraints from the string constraint specification.

Figure 2-10 shows sample code for the satisfiability-based string constraint solving procedure. The primitive search query on Line #4 is $\exists s.\textsc{wellconstrained}(r, X, s_{1,\ldots,i})$, where s represents a free variable over strings of length i, and WELLCONSTRAINED is used as in the brute-force approach to test if the string s satisfies the regular expression constraint r and the substring constraints X. The call to Sat (Line #4) returns a result indicating whether there exists any value for s satisfying the constraints. If the constraints are satisfiable, the call to Witness (Line #5) returns a value that satisfies the constraints.

In principle the satisfiability-based approach benefits from the same decoupling of search and constraints as the brute-force approach: the procedure in Figure 2-10 uses the same WELLCONSTRAINED procedure as the brute-force approach to describe the string constraints.

As with the brute-force approach to search, the satisfiability-based search can be reorganized, for instance, to search the space of strings from the language of the regular expression r, rather than strings of a fixed length (see Section 2.2.4). It is more cumbersome, however, because the primitive search procedure, $\exists x.\, \phi(x)$, assumes the variable x is entirely unconstrained, instead of, say, a string variable that is partially

```
procedure STRSOLVE(l, h, r, X)
    if Sat(∃c_r. WELLCONSTRAINED(X, s(c))) then
        return s(Witness(c_r))
    end if
    return NoSuchString
end procedure
```

Figure 2-11: Alternative satisfiability-based string constraint solver.

constrained according to the regular expression r.

To use the primitive search procedure for partially constrained variables, we may separate the specification of the string search space into two parts: a *control* c, and a function, call it s, that determines which string a control refers to. For example, if the regular expression r was `edu|net`, the control c could be a single boolean variable, and the function s is defined as:

$$s(c) = \begin{cases} \texttt{"edu"} & \text{if } c = \texttt{true} \\ \texttt{"net"} & \text{if } c = \texttt{false} \end{cases}$$

This way a query can be posed in which the variable c is entirely unconstrained, while preserving the structure of the string search space. A high-level view of how the string constraint solver would work using this technique is given in Figure 2-11. The variable c_r represents a control variable for the regular expression r.

Using control variables in this manner leads to an interesting interpretation of the satisfiability-based approach to search. The control variable c represents the choice of which alternatives to take and which substrings to use when constructing a string from a regular expression. The task of the primitive search procedure is to identify decisions for the choices that lead to a correct final result. In effect, the primitive search acts as an oracle for the choices in an otherwise nondeterministic computation. We will revisit this idea later on.

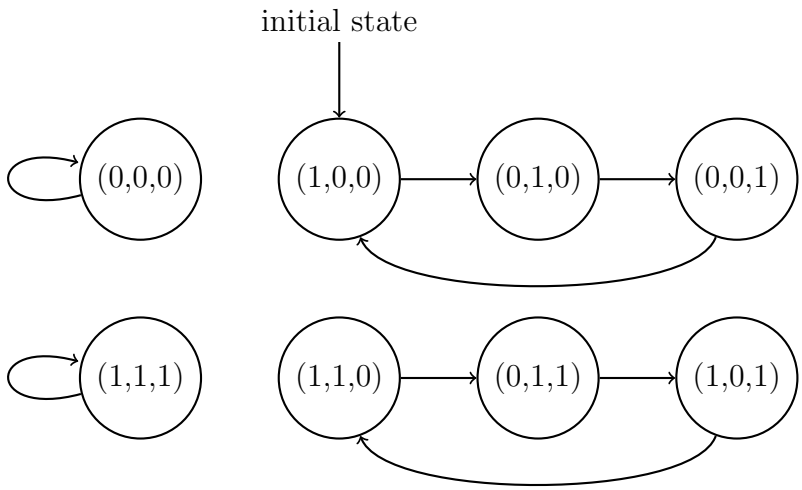

Figure 2-12: State transition diagram for a 3-bit ring counter.

2.4.2 Model Checking

The goal of model checking is to verify a hardware or software system with respect to some user-specified properties. The verification problem can be posed as a search problem: find a sequence of state transitions of the system demonstrating violation of the user-specified properties.

For example, Figure 2-12 shows the state transition diagram for a 3-bit ring counter. Initially the first bit of the counter is high. The bits of the counter are shifted to the right each cycle in circular fashion, with the rightmost bit taking the place of the leftmost. An important property of the ring counter is that at any time exactly one bit should be high. This property can be verified using a model checker that verifies the property holds for all states of the system reachable from an initial state.

Using a similar approach as in the string constraint solver, a model checker can be implemented by posing a sequence of queries for fixed-length counter-examples of increasing length. Figure 2-13 shows a bounded model checking procedure for verifying that a model M, with initial states described by the predicate I, is safe with respect to a property P for traces up to length k. The PSAFE procedure returns True if the model is safe, otherwise it returns a sequence of states that violates the property P.

```
1: procedure PSAFE(M, I, P, k)
2:     i ← 0
3:     while i < k do
4:         if Sat(∃s. I(s_0) ∧ path(M, s_{0,...,i}) ∧ ¬P(s_i))) then
5:             return Witness(s)
6:         end if
7:         i ← i + 1
8:     end while
9:     return True
10: end procedure
```

Figure 2-13: Bounded model checking procedure.

For the ring counter-example, the model M is the state transition diagram from Figure 2-12, the predicate I holds for only the initial state (1,0,0), and the property P holds for those states where exactly one bit is high. The query on Line #4 searches for a sequence of states $s = s_0, s_1, \ldots, s_i$ satisfying the following properties:

1. The first state of the sequence is an initial state. This is denoted by $I(s_0)$.

2. There exists a state transition in the model of the system between each state in the sequence and its successor state. This is denoted by $path(M, s_{0,\ldots,i})$.

3. The last state of the sequence does not satisfy the desired property of the system. This is denoted by $\neg P(s_i)$.

If the primitive search procedure finds a sequence of states satisfying these three constraints, that sequence of states is a counter-example illustrating there is a bug in the system.

In bounded model checking, the user must supply a bound k for the length of counter-examples. The PSAFE procedure verifies there are no bugs in the model that are reached within k state transitions, but the procedure cannot determine if there are bugs in the model that occur only after k state transitions. Typically there exists a model-dependent upper bound that can be used for k to ensure a model considered safe by the PSAFE procedure has no bugs for any length sequence of states.

Sheeran, Singh, and Stålmarck have shown that primitive search queries can be used not only to search for fixed-length counter-examples, but also to automatically

```
1: procedure PSAFE(M, I, P)
2:     i ← 0
3:     while i < ∞ do
4:         if Sat(∃s. I(s_0) ∧ path(M, s_{0,...,i}) ∧ ¬P(s_i))) then
5:             return Witness(s)
6:         end if
7:         i ← i + 1
8:         if not Sat(∃s. I(s_0) ∧ loopFree(s_{0,...,i}))
9:         or not Sat(∃s. loopFree(s_{0,...,i}) ∧ ¬P(s_i))) then
10:            return True
11:        end if
12:    end while
13: end procedure
```

Figure 2-14: Induction-based model checking procedure.

determine an upper bound for the argument k that ensures there are no bugs for any length sequence of states [48]. The key idea is that only sequences of states without loops need to be checked for counter-examples, because removing the loops from a counter-example results in a shorter sequence of states that exhibits the same bug. Also, if there are no sequences of states of length i that are loop free, then there are no sequences of states of any length greater than i that are loop free. A primitive search query can be used to search for a loop-free sequence of states of length i; if there are no such sequences, then there are no loop-free sequences of length i or greater, and the search procedure can terminate.

The following query determines whether there exists a sequences of states longer than length i that is loop-free:

$$\exists s.\, loopFree(s_{0,\ldots,i})$$

In the query, $loopFree(s)$ means s contains no repeated state.

A refinement of this query is to test whether there exists a loop-free sequence of states longer than length i that begins in an initial state. If there is no such sequence, the model has been completely verified. Similarly, if there exists no loop-free sequence of states longer than length i that ends in a bad state, the model has been completely verified. A revised implementation of the PSAFE procedure is shown in Figure 2-14

that automatically determines when the model has been verified rather than relying on a user-provided bound k. The queries on Line #8 and Line #9 are both used to test if the model has been completely verified, or if longer sequences of states must be considered.

In addition to the basic algorithm in Figure 2-14, Sheeran, Singh, and Stålmarck present a handful of interesting variations on the algorithm. For example, one of the variations makes it clear the algorithm uses an inductive approach to verify there are no counter-examples of any length. In another of the variations, the iteration need not start with zero-length traces, which can improve the runtime of the model checker substantially in practice. Modularity, simplicity, and flexibility in the implementation of the model checker would make it easier to explore the performance impacts of the different variations.

2.4.3 Program Synthesis

Program synthesis searches for snippets of code to complete a partially implemented program in a way that satisfies the program's specification. For example, Figure 2-15 shows an unoptimized specification and a sketch of an optimized function for isolating the rightmost unset bit of a word. The `isolate0` function returns a word with a single bit set at the position where the rightmost unset bit occurs in the input word. These functions are expressed using Sketch [50], a C-like language for program synthesis. Each token `??` represents a hole for the synthesizer to fill in. In this case the program synthesizer will complete the sketch using 0 for the first hole, and 1 for the second:

```
bit[W] i0sketch(bit[W] x) {
  return ~(x + 0) & (x + 1);
}
```

Solar-Lezama has shown a clever way of decomposing the program synthesis problem into primitive search queries, called counter-example guided inductive synthesis (CEGIS) [50]. It is based on the idea that for programs, a small number of inputs exercise all of the interesting cases. Primitive search queries can be used both to find

```
bit[W] isolate0 (bit[W] x) {
  bit[W] ret=0;
  for (int i = 0; i < W; i++) {
     if (!x[i]) {
        ret[i] = 1;
        break;
     }
  }
  return ret;
}

bit[W] i0sketch(bit[W] x) implements isolate0 {
  return ~(x + ??) & (x + ??);
}
```

Figure 2-15: Isolate0 specification and sketch for program synthesis.

a candidate program that satisfies the specification on a small number of interesting inputs, and also to search for new interesting inputs for which the candidate program fails to meet the specification.

Figure 2-16 shows the CEGIS procedure. The query on Line #4 searches for program snippets to complete the sketch S so that the sketch satisfies the specification, P, on all interesting inputs x^*. The program snippets are represented using the variable c, and the interesting inputs are those inputs collected so far in the set I. The result of this search is a candidate solution, c^*. (We use c to refer to an unconstrained variable in the query, and c^* to refer to the concrete candidate solution if the query is satisfiable.) If no candidate solution can be found, there is a bug in the user's program sketch.

The query on Line #9 is used to verify the candidate solution c^* by searching for a counter-example input x on which the result of the synthesized program differs from the specification. If there is no such input, then the candidate solution c^* is valid for all inputs, and synthesis is complete. Otherwise, the concrete input x is an interesting input and is added to the collection I.

Note that the CEGIS procedure describes the search for a program in terms of a control variable c and a mapping S from control to program. This is analogous

```
1: procedure CEGIS(P, S)
2:     I ← ∅
3:     while True do
4:         if Sat(∃c. ⋀_{x*∈I} P(x*) = S(x*, c)) then
5:             c* ← Witness(c)
6:         else
7:             return BuggySketch
8:         end if
9:         if Sat(∃x. P(x) ≠ S(x, c*)) then
10:            I ← I ∪ {Witness(x)}
11:        else
12:            return c*
13:        end if
14:    end while
15: end procedure
```

Figure 2-16: Procedure for counter-example guided inductive synthesis.

to how we used a control variable c_r and mapping s from control to string in the alternate organization of the string constraint solver.

2.5 Satisfiability Solvers

Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers are procedures that can be used for solving primitive search queries in satisfiability-based search. The reason the satisfiability-based approach to search has been applied to so many problems in the last decade is because SAT and SMT solvers have matured enough to solve practically sized search queries efficiently.

Unfortunately, SAT and SMT solvers do not support general purpose programming features for describing constraints. In this section we discuss what features SAT and SMT solvers do support. In the next section we will discuss how the limited features for describing constraints supported by SAT and SMT can have a substantial impact on the modularity, simplicity, and flexibility of the satisfiability-based approach to search in practice.

$$
\begin{array}{lrcl}
 & v & \in & \textit{Boolean Variable} \\
\text{Formula} & \phi & ::= & \texttt{true} \mid \texttt{false} \mid v \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2
\end{array}
$$

Figure 2-17: Syntax of boolean formulas.

2.5.1 Satisfiability

The problem of Satisfiability is to determine if there exists an assignment to variables of a boolean formula under which the formula is satisfied. In terms of the primitive search query, $\exists x.\, \phi(x)$, x is a finite collection of boolean variables, and $\phi(x)$ is expressed as a boolean formula. The syntax for a boolean formula is shown in Figure 2-17. Boolean formulas are made up of boolean constants `true` and `false`, boolean variables, and the logical connectives NOT ($\neg$), AND ($\wedge$), and OR ($\vee$).

Example 2.1 A Boolean Formula**.**

$$\phi_{2.1} = (v \vee w \vee \neg z) \wedge z \wedge (y \vee \neg w)$$

Example 2.1 shows a boolean formula with four variables: v, w, y, and z. The problem of satisfiability for this formula is to determine whether there exists an assignment of boolean values to the variables v, w, y, and z under which $\phi_{2.1}$ is `true`. Mathematically this is expressed as the query:

$$\exists v, w, y, z.\, \phi_{2.1}(v, w, y, z) = \texttt{true}$$

The formula $\phi_{2.1}$ is satisfiable, because when $v = \texttt{true}$, $w = \texttt{false}$, $y = \texttt{true}$, and $z = \texttt{true}$, $\phi_{2.1}(v, w, y, z)$ evaluates to `true`. We call the assignment $\{v = \texttt{true}$, $w = \texttt{false}$, $y = \texttt{true}$, $z = \texttt{true}\}$ a satisfying assignment. For the particular boolean formula $\phi_{2.1}$ shown here, there are multiple satisfying assignments. A formula is *unsatisfiable* if it has no satisfying assignments.

A SAT solver computes whether a boolean formula is satisfiable or not. If the formula is satisfiable, the solver returns a satisfying assignment, otherwise the solver indicates the formula is unsatisfiable.

Though the problem of satisfiability is NP-complete [13], SAT solvers leverage

efficient heuristic search algorithms based on the Davis-Putnam-Logemann-Lovland (DPLL) [15, 16] procedure for satisfiability that scale well for many practical problem instances due to decades of research and implementation efforts.

SAT solvers support queries with only a finite number of variables, where each variable is a boolean variable. The constraints must be described using a boolean formula. SAT solvers do not support operations on values of non-boolean types, such as integers, nor do they support procedure calls, macros, loops, program variables, control flow, or other features typical of a programming language in the specification of constraints.

2.5.2 Satisfiability Modulo Theories

Satisfiability Modulo Theories is an extension to the problem of Satisfiability that augments the syntax of formulas [3]. For example, an SMT solver with the background theory of linear integer arithmetic accepts formulas containing integer variables, integer literals, and basic integer operations such as addition, subtraction, equality, and comparison operators.

Example 2.2 An SMT Formula**.**

$$\phi_{2.2} = (x \vee (y + w < z)) \wedge (z = 5) \wedge (y \neq w)$$

Example 2.2 shows an example of an SMT formula with a single boolean variable x, and three integer variables, w, y, and z. The SMT formula $\phi_{2.2}$ is satisfiable under the assignment $\{x = \texttt{true}, y = 0, w = 1, z = 5\}$.

Common background theories supported by SMT solvers include theories of integers, bit-vectors, real numbers, and arrays. SMT solvers can leverage higher-level information to efficiently solve the formula, such as commutativity and associativity of the operator `+`.

The features supported by SMT solvers for programming abstractions, encapsulation, and modularity vary enormously by solver. All SMT solvers support some notion of a `let` construct to define intermediate variables. The following example

shows an SMT formula using `let` to explicitly express sharing in the formula:

$$\phi(x) = \left(\begin{array}{l} \texttt{let } \texttt{y} = \texttt{x} + \texttt{x} + \texttt{x} + \texttt{x} \\ \quad\quad \texttt{z} = \texttt{y} + \texttt{y} + \texttt{y} + \texttt{y} \\ \texttt{in } \texttt{z} < 0 \end{array} \right)$$

Some SMT solvers concentrate on making one or a few background theories work together efficiently, without providing any support for other programming abstractions. For example the STP [22] solver supports bit-vectors and arrays very efficiently, but aside from primitive operations related to bit-vectors, arrays, and the core SMT theory, no other programming abstractions are provided. Other SMT solvers, such as Yices1 [18] and Z3 [17], support a large number of background theories, and attempt to provide user-level abstractions that simplify use of the solver. They include some support for using tuples, user-defined record types, and top-level declarations of terms and functions, which the solver will fully inline into the constraints for the user. Typically SMT solvers do not support the array of features available in a modern object-oriented or functional programming language, such as package management, module support, function overloading, abstract datatypes, metaprogramming, loops and recursion, input/output operations, or manipulation of runtime state. There tends to be no support in the language of the solvers for programmatically allocating new variables or constructing constraints. It is generally assumed that SMT-based tools will be developed using a separate host language to construct complex queries.

2.6 Challenges of Using Satisfiability-Based Search

Though SAT and SMT solvers make the satisfiability-based approach to combinatorial search viable from a performance standpoint, in practice, because they have limited support for describing constraints using general purpose programming features, developing an application that effectively uses a SAT or SMT solver can be a daunting task.

For a sense of the magnitude of effort required to develop a practical tool based on SAT or SMT, consider the number of lines of code used to implement practical versions of the string constraint solver, model checker, and program synthesis tools:

- Our string constraint solver example is based on HAMPI [31], a satisfiability-based string constraint solver implemented in 20,000 lines of Java code.
- A reference implementation of a satisfiability-based model checker for the annual hardware model checking competition is implemented in 14,000 lines of C code.
- The authors of the CEGIS algorithm for program synthesis developed the Sketch tool based on that algorithm using a mix of 85,000 lines of Java and 20,000 lines of C code.

These large code bases require a substantial investment of developer effort to create, debug, optimize, and maintain.

Why are such large code bases required for satisfiability-based applications using SAT and SMT solvers? The following two anecdotes from developers of SMT-based tools hint at the reasons.

Dave et al. developed a fully automated SMT-based tool for verifying the correctness of microarchitectural refinements in rule-based systems[14]. This rule-based refinement tool searches for execution traces that can occur in an implementation of the system that are invalid in the specification of the system. The developers of this tool describe future improvements they would make if not for the high engineering costs involved:

1. Leverage theories of FIFOs, arrays, and uninterpreted functions instead of just the theory of bit-vectors to dramatically reduce the complexity of the SMT queries.
2. Use a more efficient interface to the SMT solver than one based on file-level IO, as more than half the compute time comes from marshaling and demarshaling the query representation.

KLEE [8] is an SMT-based tool for automated generation of test cases for systems programs. The dominating factor in the runtime of the KLEE tool was the evaluation of the SMT queries. To make KLEE effective, the developers spent significant effort to optimize the SMT queries generated. The optimizations they implemented include expression rewriting, constraint set simplification, implied value concretization, leveraging constraint independence and using a counter-example cache. These optimizations reduced the amount of runtime dedicated to SMT solving from 92% of the tool's runtime to 41%.

The satisfiability-based approach to search using SAT and SMT solvers is not as modular, simple, and flexible in practice as it is in theory because user-level constraints must be transformed into constraints supported by the SAT and SMT solvers. In the remainder of this section we will show that this transformation process is complex to implement, interfacing with SAT and SMT solvers is nontrivial, and the developer must optimize the transformation process before they can take advantage of the performance of SAT and SMT solvers for solving queries.

2.6.1 Transforming User-Level Constraints to SAT and SMT

The procedures we presented in Section 2.4 for string constraint solving, model checking, and program synthesis using the satisfiability-based approach to search are missing an important component required to work with SAT or SMT solvers: the user-level data types and constraints must be transformed into SAT or SMT-level variables and formulas.

For example, for the string constraint solver, we wrote:

$$\text{Sat}(\exists s.\, \textsc{wellconstrained}(r, X, s_{1,\ldots,i}))$$

This describes a satisfiability query with a free variable s, where s is a string of i characters, and a formula $\textsc{wellconstrained}(r, X, s_{1,\ldots,i})$, which says the string s must satisfy the regular expression constraint r and all of the substring constraints X. Neither SAT nor SMT solvers support variables that are strings of a fixed length,

however, and WELLCONSTRAINED$(r, X, s_{1,\ldots,i})$ is not a valid term to use in a SAT or SMT formula.

The developer must supply a procedure that transforms the high-level query to queries supported directly by the solvers. As with the implementation of WELLCONSTRAINED for the brute-force search approach, this transformation procedure depends on domain details, such as what alphabet to use for characters of the string, what syntax of regular expressions is supported, and what precisely it means for a regular expression to match a string. For satisfiability-based search, this procedure also depends on how user-level variables and constraints are encoded as SMT variables and formulas. For example, a character could be encoded using a bit-vector variable, integer variable, or a collection of boolean variables.

For a sense of the complexity of the transformation procedure, consider how the CONTAINS procedure in Figure 2-3 for testing if a string contains a given substring must be modified to use an SMT solver in practice.

First we must choose how to encode a fixed-length string variable using SMT variables. For instance, a string of length i can be encoded using i different bit-vector variables, one for each character of the string. Instead of operating on a string s and a substring x, the CONTAINS procedure must know which SMT variables are being used to represent the characters of the string s. In place of the argument s, the procedure is modified to accept integer arguments l and h, which are the variable ids for the first and last characters in the string respectively. Instead of returning a simple value True or False, the procedure is modified to return an SMT formula.

Figure 2-18 shows pseudocode for the modified CONTAINS procedure. Instead of searching for the first position of the string s for which x is a prefix, the CONTAINS procedure (Line #1) now constructs the formula f, calling the procedure PREFIX on Line #7 to construct a sub-formula for all the positions in the string s.

The PREFIX procedure (Line #13) has been modified similarly. It iterates through each character of the string x, constructing a formula f that is satisfied only when each character of the string x is equal to the corresponding character of s. The characters of x are user-level characters, and must be encoded using the ENCODE procedure

```
1: procedure CONTAINS(x, l, h)
2:     if x = ϵ then
3:         return true
4:     end if
5:     f ← false
6:     while l ≤ h do
7:         f ← f ∨ PREFIX(x, l, h)
8:         l ← l + 1
9:     end while
10:    return f
11: end procedure
12:
13: procedure PREFIX(x, l, h)
14:    if |x| > h − l then
15:        return false
16:    end if
17:    f ← true
18:    i ← 1
19:    while i ≤ |x| do
20:        f ← f ∧ (v_l = ENCODE(x_i))
21:        i ← i + 1
22:        l ← l + 1
23:    end while
24:    return f
25: end procedure
```

Figure 2-18: Procedure for testing a substring constraint.

to be used in the constructed formula (Line #20), where they are compared to the SMT-level representation of the characters of s.

This transformation procedure is more complex, less modular, and less flexible than the original CONTAINS procedure used for brute-force search:

- The original CONTAINS procedure was modified to produce this procedure. It is not possible to reuse a library procedure for this constraint unless the library procedure is meant for use specifically with SMT solvers. The CONTAINS procedure shown here is simple enough that a library procedure is not required. For the regular expression constraint, however, if the developer cannot reuse a library procedure, then they are responsible for developing their own regular expression match algorithm; they cannot reuse a highly-tuned library.

- The implementation is expressed in terms of l and h, which represent an encoding of the string s using SMT variables. It does not operate directly on a string s. This indirection increases the cognitive burden of understanding the code.

- The procedure assumes bit-vector variables v_l through v_h are used to represent the string s. If the designer wants to use a different encoding, or a different scheme for naming variables, the procedure must be modified. For example, if the designer wants to reorganize the search procedure to search the space of strings by the regular expression constraint instead of the length constraint, that requires modification of the CONTAINS procedure. This was not the case for the brute-force approach to search, and in principle should not need to be the case for the satisfiability-based approach either.

- If x is the empty string, the CONTAINS procedure can immediately return the result `true` without consulting the SMT variables used to represent the string s. Note that the procedure returns the SMT formula `true` (Line #3), not the simple value True. In this case, the procedure for constructing the formula has enough information to produce a result directly. This can be important for optimizations, which we discuss in more depth in Section 2.6.3.

```
procedure INTERPRET(i)
    s ← ϵ
    while i > 0 do
        a ←DECODE(Witness(v_i))
        s ← as
        i ← i − 1
    end while
    return s
end procedure
```

Figure 2-19: Procedure to interpret string result from SMT solver.

- If there is a bug in the CONTAINS procedure, it could either be a bug in the process of constructing the formula f, or a bug in the generated formula f. This makes it more complicated to track down bugs, especially because the only indication of the bug may be that the SMT solver returns an answer that the developer knows must be wrong. If the bug is in the generated formula, it could be difficult to locate given how large the constructed formula can be, and once located, it can be difficult to determine which part of the procedure constructed the faulty part of the formula.

For a working SMT-based implementation, the procedures for testing each kind of string constraint must be modified like the CONTAINS procedure was. The developer must also provide a procedure to interpret the result of the SMT solver. For example, the satisfying assignment $\{v_1 = 102, v_2 = 111, v_3 = 111\}$ corresponds to the string `"foo"`. A sample procedure for interpreting the satisfying assignment of the string constraint solver's query is given in Figure 2-19. The INTERPRET procedure iterates in reverse order over the SMT variables used to encode the string. It calls Witness to retrieve the value of each variable, and it calls DECODE to convert the bit-vector literal to the corresponding character it represents. The decoded characters are concatenated together to produce the string s.

Figure 2-20 shows the string constraint solver modified to use SMT. Instead of a query involving a string variable s, the query is expressed in terms of bit-vector variables labeled v_1 through v_i. The WELLCONSTRAINED procedure operates on the

```
procedure STRSOLVE(l, h, r, X)
    i ← l
    while i ≤ h do
        if Sat(∃v_1, ..., v_i. WELLCONSTRAINED(r, X, 1, i) then
            return INTERPRET(i)
        end if
        i ← i + 1
    end while
    return NoSuchString
end procedure
```

Figure 2-20: Revised satisfiability-based string constraint solver.

identifiers for these variables, 1 through i, instead of directly on the string the variables represent. Though the implementation of STRSOLVE does not look too different from the high-level procedure given in Section 2.4 on the surface, this implementation is much less generic than the high-level satisfiability-based implementation, because it assumes a specific encoding for the string.

How hard is it to reorganize the STRSOLVE procedure to search for strings by the regular expression r instead of by length? Generating queries involving control variables increases the complexity of transforming user-level constraints to SMT constraints: it is simpler to transform the user-level notion of a string of characters to the SMT notion of a collection of bit vector variables representing characters than to transform the user-level notion of strings constructed by a regular expression to the collection of boolean variables that control which alternatives in the regular expression were used to generate the string.

2.6.2 Solver Interface Challenges

Another challenge of using SAT and SMT solvers in practice is that the developer must write code to transmit the constructed query to the solver and collect the results from the solver after the query has been solved. We have used the functions Sat and Witness to represent this process in our example procedures. In practice the developer must provide their own implementation of Sat and Witness, and this implementation is specific to the SAT or SMT solver the developer plans to use.

The interface challenges for using SAT solvers differ from those for SMT solvers. SAT solvers have simple, standardized interfaces, but the process of converting formulas to conjunctive normal form required for the SAT interfaces can have a significant impact on how long it takes to solve the query. SMT solvers have complex, nonstandard interfaces that make it difficult to experiment with different solvers and ways of encoding the user-level variables constraints as SMT variables and constraints.

The SAT Interface

The overwhelming majority of SAT solvers expect the boolean formula of the SAT query to be in conjunctive normal form (CNF). Conjunctive normal form expresses a boolean formula as the conjunction of clauses, each of which is a disjunction of variables or their negations.

Example 2.3 A Boolean Formula in Conjunctive Normal Form**.**

$$\phi_{2.3} = (v \vee w \vee \neg z) \wedge z \wedge (y \vee \neg w)$$

Example 2.3 shows an example of a boolean formula in conjunctive normal form.

Example 2.4 A Boolean Formula Not in Conjunctive Normal Form**.**

$$\phi_{2.4} = (v \wedge (y \vee \neg w) \wedge z) \vee (y \wedge w \wedge z)$$

Example 2.4 shows a boolean formula equivalent to $\phi_{2.3}$ that is not in conjunctive normal form because it includes a disjunction of terms that have conjunctions.

CNF was popularized by the Second DIMACS Implementation Challenge [30] and the Davis-Putnam-Logemann-Lovland (DPLL) procedure for satisfiability of CNF formulas [15, 16], which most SAT solvers are based on. The interface to SAT solvers is either text-based, using the format from the DIMACS challenge, or library-based, with a small number of library functions for creating new boolean variables and constructing the formula one clause at a time.

Example 2.5 A DIMACS Formatted SAT Formula**.**

```
p cnf 4 3
1 3 -4 0
4 0
2 -3 0
```

Example 2.5 shows the CNF formula from Example 2.3 in DIMACs format. The first line indicates the formula is in CNF with four variables and three clauses. Each subsequent line describes a clause, using positive integers to identify variables, negative integers to refer to negated variables, and 0 to indicate the end of the clause.

Example 2.6 Library-Based Construction of Formula for MiniSat**.**

```
Solver s;
Var v = s.newVar();
Var w = s.newVar();
Var y = s.newVar();
Var z = s.newVar();
s.addClause(mkLit(v,true), mkLit(w, true), mkLit(z, false));
s.addClause(mkLit(z,true));
s.addClause(mkLit(y,true), mkLit(w, false));
```

Example 2.6 shows C++ code using the library interface to the MiniSat solver to construct the CNF formula from Example 2.3.

The main challenge of interfacing with a SAT solver is converting the boolean formula to CNF form. Converting a boolean formula into an equivalent formula in conjunctive normal form can incur exponential growth in the size of the formula. Using the approach of Tseitin [57], it is possible to construct a formula in conjunctive normal form with only linear growth in the size of the formula, when the constructed formula is equi-satisfiable to the original formula rather than equivalent. An equi-satisfiable formula is satisfiable if and only if the original formula is satisfiable, but it may include additional variables not present in the original formula.

Chambers, et al. show that the details of the conversion process can have a substantial impact on solving time, and they suggest significantly more advanced CNF generation techniques than that of Tseitin [9]. These techniques rely on more complicated interfaces to convey and preserve high level information in the original formula.

The SMT Interface

The interface to SMT solvers is much more complicated than the interface to SAT solvers, because SMT solvers support a much larger syntax of formulas. As with SAT solvers, SMT solvers have both text-based and library-based interfaces.

Though there has been some standardization of the text-based format for SMT queries, most notably SMTLIB [4], this interface supports only the lowest common denominator among SMT solvers, and does not provide a standard format for describing satisfying assignments. The primary text-based interface for each solver is the solver's own custom format for expressing SMT queries, which, unlike the SMTLIB interface, provides access to the special features of the solver.

There is no standardization of the library-based interface to SMT solvers. The library-based interfaces are complicated. For example, the STP [22] solver's C-interface defines seven different data types and exports 125 different functions. There are functions for configuring the solver, working with variable types, constructing nodes of a formula, printing formulas, accessing counter-examples, controlling memory allocation and deallocation, and handling errors. STP is one of the simpler SMT solvers, because it includes only the background theories of bit-vectors and arrays, and not integers or other high level constructs such as functions and records.

Understanding how best to interface with an SMT solver is not a trivial task. For example, STP's interface includes at least five different functions for creating a bit-vector constant. Which of these function should be used? Are there performance consequences for choosing one over the other?

A consequence of the complexity of the SMT solver interface is that a nontrivial amount of development effort must be devoted to working with and tuning the inter-

action with the solver. Because solver interfaces are nonstandard and solvers support different background theories, adapting an SMT-based application to use a different solver or background theory can require reimplementation of substantial parts of the application. It is not always easy to take advantage of the latest and greatest solvers, and libraries for generating SMT queries can be reused only in applications using the specific solver supported by the libraries.

2.6.3 Optimized Transformation of User-Level Constraints

For an implementation of a satisfiability-based application to run, it must include all of the pieces described so far:

- Decomposition of the combinatorial search problem into SAT queries, such as the decompositions described in Section 2.4 for string solving, model checking, and program synthesis.
- A procedure for constructing the actual SAT or SMT variables and formula corresponding to a high-level SAT-like query, such as the modified CONTAINS procedure of Figure 2-18 and the INTERPRET procedure of Figure 2-19 described in Section 2.6.1.
- Code to drive the SAT or SMT solver interface as described in Section 2.6.2.
- Routines for interacting with the user, such as a command line parser, a parser for string constraint specifications, and a routine to output the result of the tool to the user.

In theory, an implementation with all of these components is functional. In practice, there will likely be performance problems so severe as to make the implementation unusable on any but the most trivial problems. The basic implementation may take hours to run, or, more likely, exhaust all of the available memory on the machine.

The three most likely sources of the performance problems are:

1. The decomposition of the high-level problem into SAT queries does not work well. For the developer, this is the most interesting performance problem to

remedy, because the solution is tied to the specific combinatorial search problem they are solving. If the satisfiability-based approach had more modularity, simplicity, and flexibility in practice, it would be much easier for a developer to experiment with different decompositions.

2. The time for the SAT solver to solve the query dominates. Given how well-tuned SAT solvers are, this problem likely indicates the SAT query being posed is fundamentally hard. A developer can either wait for SAT solvers to improve, or restructure their decomposition to avoid these hard queries.

3. The time for transforming user-level variables and constraints to SAT or SMT variables and constraints dominates, either because there is exponential growth in the size of the query, or because the query is not properly bounded to finite size.

The third source of performance problems is troubling, because one of the primary motivations for using satisfiability-based search is that the primitive search procedure is optimized using advanced techniques from a community of experts to handle challenges of exponential growth. In practice, however, the developer for a particular application cannot entirely rely on the SAT or SMT solver to handle the challenges of combinatorial growth. The reason for this is SAT and SMT solvers do not have access to the user-level constraints. Using a SAT or SMT solver to efficiently solve a query is of little consequence unless the user-level constraints can be efficiently transformed into a query first.

Solving performance problems in the construction of the queries is an engineering challenge. It may be that the construction procedure is fundamentally hard, and cannot scale to large input sizes. In this case, the question is how to optimize the construction procedure so that it works reasonably well on the practically sized problems the tool is most expected to be used with.

The following sections give examples of what may cause the construction of a formula to explode, and how these issues can be addressed in the implementation of the formula construction procedure.

Preservation of Sharing

Example 2.7.

```
let  y = x + x + x + x
     z = y + y + y + y
in z < 0
```

Example 2.7 shows a contrived domain-level constraint that illustrates the importance of preservation of sharing when constructing the query. The variable y is defined as four repeated additions of the free integer variable x, z is defined as four repeated additions of y, and the constraint is that z should be less than 0.

Example 2.8.

$$((x + x + x + x) + (x + x + x + x) + (x + x + x + x) + (x + x + x + x)) < 0$$

Example 2.8 shows how a simple constraint at the domain level (Example 2.7) can lead to an exponentially larger constraint at the level of an SMT query if the construction of the query does not preserve sharing.

If you were to manually construct an SMT query for the domain-level query in Example 2.7, you would likely not inline the intermediate variables as in Example 2.8. When the construction process is automatic and domain-level queries are more complicated, however, it can be difficult to ensure sharing is preserved. Note that though the sharing in Example 2.8 is clear in the static structure of the query, often the most important sharing to preserve is available only dynamically. For example, the implementation of regular expression match, depending on the specific regular expression, can be memoized to avoid duplicated matching of parts of the regular expression against substrings. For the domain-level optimization of memoization in the regular expression match to be of any use, that sharing must be preserved in the generated SMT query.

Simplifications

Often straight-forward simplifications of formulas can drastically reduce their size.

Example 2.9.

$$\begin{aligned}
&\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"ababab"} \ \vee \\
&\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"ababcd"} \ \vee \\
&\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"abcdcd"} \ \vee \\
&\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"cdcdcd"}
\end{aligned}$$

$$\begin{aligned}
&(97 = 97 \wedge 98 = 98 \wedge x_1 = 97 \wedge x_2 = 98 \wedge 99 = 97 \wedge 100 = 98) \ \vee \\
&(97 = 97 \wedge 98 = 98 \wedge x_1 = 97 \wedge x_2 = 98 \wedge 99 = 99 \wedge 100 = 100) \ \vee \\
&(97 = 97 \wedge 98 = 98 \wedge x_1 = 99 \wedge x_2 = 100 \wedge 99 = 99 \wedge 100 = 100) \ \vee \\
&(97 = 99 \wedge 98 = 100 \wedge x_1 = 99 \wedge x_2 = 100 \wedge 99 = 99 \wedge 100 = 100)
\end{aligned}$$

Example 2.9 shows a domain-level query involving strings and a straight-forward encoding of that query as an SMT query using integer variables and ASCII encoding to represent the domain-level characters.

This query can be simplified by recognizing that $97 = 99$ is `false`, and because of that, the entire last clause must be `false`. Those simplifications, along with other simple simplifications can reduce the size of the original query to a much smaller equivalent query:

$$(x_1 = 97 \wedge x_2 = 98) \vee (x_1 = 99 \wedge x_2 = 100)$$

Employing these simple simplifications during the construction of a query can lead to much smaller queries, making otherwise impractically large queries practical to construct. These simplifications will also make the query easier for the SAT or SMT solver to solve, though presumably the SAT or SMT solver would perform these simplifications anyway, so the time to solve the query will not be reduced substantially by simplifying the query ahead of time.

Pruning

Example 2.10.

```
let factorial n =
        if n < 2
           then 1
           else n * factorial (n - 1)
in x > 0 && x < 10 && factorial x > 200
```

Example 2.10 shows a contrived domain-level constraint that looks for an integer x between 0 and 10 with factorial greater than 200. SAT and SMT solvers do not support constraints involving unbounded recursion. In order to generate a SAT or SMT formula encoding this domain-level constraint, the factorial function must be inlined. An automatic procedure for inlining the factorial function must ensure it is inlined a finite number of times, otherwise the generated query will be infinitely large.

In this example, because of the constraint that the variable x is between 0 and 10, the bound on how far the factorial function is unrolled could be identified during query construction; eventually the argument n to factorial, though not known concretely, will for certain be less than 2, and the recursive call can be pruned away.

2.6.4 Consequences for Users of SAT and SMT

The limited support SAT and SMT solvers have for describing constraints using general purpose programming features has significant consequences for users of SAT and SMT solvers when developing satisfiability-based applications. An immediate consequence is that satisfiability-based search applications are hard to create, debug, optimize, and maintain. Long term consequences are:

- The startup costs for using the satisfiability-based approach may be so high that, even with a good insight for how to decompose a combinatorial search problem into primitive search queries, it may not be practically viable to implement.

- Limited modularity, abstraction, and encapsulation hinders design exploration, making it hard to gain new insight by experience.

- Code reuse across applications is discouraged, because the code is specialized for a particular application, solver, encoding, and usage scenario.

The satisfiability-based approach to combinatorial search problems is valuable, because it makes it possible to think about how to solve these problems in a practically viable way. A developer is still likely to get further using a satisfiability-based approach than by developing a fully custom search procedure. To realize the full benefits of the satisfiability-based approach, however, these practical limitations must be addressed.

2.7 Smten: Enhanced Satisfiability-Based Search

This thesis is about overcoming the practical limitations of the satisfiability-based approach to developing combinatorial search problems. The goal is to significantly reduce the costs required to develop practical satisfiability-based applications by improving the modularity, simplicity, and flexibility of the satisfiability-based approach to search.

The need to transform user-level variables and constraints to SAT or SMT variables and constraints contributes the most to the reduced modularity, simplicity, and flexibility of the satisfiability-based approach to search. We propose an enhancement to the functionality supplied by SAT and SMT solvers for primitive search called Smten search. Smten search overcomes the practical limitations of satisfiability-based search by directly supporting user-level search queries with a simple interface integrated into a general-purpose language for developing combinatorial search applications.

Smten search is made possible using a functional language, in our case Haskell, which is capable of expressing general-purpose computation in a form that can be used for SAT and SMT constraints. By supporting general-purpose computation in

String Constraint Solver		Model Checker		Program Synthesis	
20,000	lines Java	14,000	lines C	105,000	lines Java and C
1,000	lines Smten	400	lines Smten	3,000	lines Smten

Figure 2-21: Comparison of effort between SAT/SMT search and Smten search.

search, the developer does not have to provide a translation layer mapping user-level constraints to the supported SAT or SMT constraints, and because constraints are expressed at a high level, the burden for efficient query construction falls to the expert implementors of the Smten search procedure instead of the application developer. Additionally, because the language of constraints is the same as the language of general-purpose computation, the interface to search can be drastically simplified by reusing the syntax and features built into the language. Smten brings the modularity, simplicity, and flexibility of the brute-force approach to search to the satisfiability-based approach to search.

Figure 2-21 shows a preview of the results of this thesis, demonstrating that applications developed using Smten search require orders of magnitude fewer lines of code than traditional, hand-coded, satisfiability-based implementations. The Smten-based implementations are comparable in performance to the hand-coded implementations.

The challenges this thesis addresses in order to support Smten search are:

- Designing a simple search interface capable of expressing constraints at the domain level that supports efficient reduction to SAT and SMT.

- Integrating standard language features with SAT and SMT. The most notable feature is support for Turing-complete computation in search constraints. Other features that must be considered are input/output operations, primitive type conversions, operations from unsupported background theories, and other operations that are not directly supported in SAT or SMT formulas.

- Developing an implementation of Smten search with good practical performance, so that Smten-based applications perform as well as hand-coded SAT or SMT-based applications.

2.8 Related Work

The idea of Smten is to add a search primitive to a functional programming language that supports user-level descriptions of constraints for satisfiability-based search. In this section we review other approaches that have been taken to improve the modularity, simplicity, and flexibility of using SAT and SMT solvers for the satisfiability-based approach to search. Some of the work we discuss is not primarily intended for developing combinatorial search applications using SAT and SMT solvers, but is still relevant.

2.8.1 SMT Solver Features and Theories

The background theories of SMT solvers allow the developer to directly express their queries at a higher level than SAT solvers. SMT solvers take advantage of the higher-level description of queries to improve performance.

Commercial SMT solvers such as Microsoft's Z3 [17] and SRI International's Yices1 [18] have many features and background theories to make the SMT solvers easier to use. For example, these solvers have some support for record types and lambda terms in describing constraints. While this support allows the developer to rely more on the solver for optimizations of higher-level constructs, SMT solvers do not yet support the general purpose programming required by developers to construct a query based on a complex input specification from the user.

Continuing to add specific features and background theories to SMT solvers makes it easier to transform user-level constraints to SMT-level constraints, but to fully solve the problems due to the transformation of user-level constraints to SMT-level constraints, the SMT solvers would need to support the wide array of general purpose programming features developers rely on for describing their programs.

2.8.2 Library Language Bindings

Libraries such as the Python and F# bindings for the Z3 solver simplify interfacing with SMT solvers from general purpose languages. These libraries for the most

part are wrappers around the raw interface of the SMT solver. They do not alter or optimize the way queries are constructed. It is up to developers to implement optimizations on top of these libraries when constructing queries.

2.8.3 Domain Specific Embedded Languages for SMT

More sophisticated interfaces to SMT have been built using a Domain Specific Embedded Language (DSEL) [36] approach. Domain specific embedded languages piggyback off of a general-purpose host language to provide domain specific languages with support for standard programming abstractions. The benefits of using a DSEL are that programmers can use familiar syntax and features from the host language, they benefit from static syntax and type checking of the host language, and different DSELs using the same host language compose more easily than traditional domain specific languages.

Examples of DSELs for SMT solvers include SBV [19] and YicesPainless [53], which are both DSELs using Haskell as the host language. Scala^{Z3} [33] is an embedding for the Z3 SMT solver in Scala, and Z3.rtk [1] is an embedding for the Z3 SMT solver in Racket [20].

These DSELs provide a full metaprogramming layer to generate SMT queries. While this makes it much easier to describe and generate complex SMT queries, there is still an explicit partition between the domain-level description and the SMT-level query, and the metaprogramming layer does not provide any support for optimizing domain-level query construction. This approach is limited to optimizing the primitive types and operations at the lowest level, and fails to address a much broader scope of possible optimizations and structure in the metaprogramming layer, in particular for user defined data types.

The reason DSELs are not as well suited for the domain of SMT queries as they are for other domains is because the search procedure used by SMT solvers can benefit drastically if it has access to the structure of the metaprogram. The benefits of the DSEL approach, however, come from hiding the structure of the metaprogram from the domain specific constructs.

2.8.4 General-Purpose SMT-Based Applications

Some SMT-based tools support more general-purpose user constraints. For example, KLEE [8] is a tool for automatic generation of test cases for systems programs, and Sketch [50] is a tool for program synthesis of C-like programs. Though not originally intended for use as general purpose tools for solving combinatorial search applications, because these tools provide support for higher-level constraints such as procedure calls and user-defined data types, they are often used instead of SMT solvers for complex applications.

The goal of Smten is to provide support for completely general-purpose user constraints. With support for completely general-purpose user constraints, application developers will not be adversely affected by the restrictions and domain-specific assumptions of KLEE and Sketch. Many of the techniques and optimizations in the implementation of KLEE and Sketch will be applicable to Smten. Ideally the techniques and optimizations that are useful for a variety of SMT-based applications are implemented in one place rather than reimplemented for each SMT-based application. If Smten is successful, we hope to see KLEE and Sketch implemented using Smten.

2.8.5 Static SMT-Based Analysis of Functional Languages

SMT solvers have been leveraged in compilation of functional languages, which requires construction of queries from the syntax of a functional language. Tools such as Leon [54] and HALO [60] generate and solve queries at compile time. They are used for static analysis instead of developing tools for combinatorial search problems. For these tools, optimizations based on dynamic data are not nearly as important as for Smten. Kuncak et al. [35] statically generate queries for Scala that can depend on runtime parameters, but the structure of the query is fixed statically.

Research for these tools has provided some techniques for supporting unbounded recursion in the description of constraints. Leon supports unbounded recursion in the description of constraints by incrementally unrolling recursive function calls and using unconstrained SMT variables to represent the possible results of the recursive

calls that have not been fully unrolled. We adapt this approach for Smten to work efficiently when dynamic behavior and high performance are more critical.

2.8.6 Integration of Functional Languages and SMT

Kaplan [34] and Rosette [56] are both intended for development of applications that execute SAT and SMT queries dynamically. They are the work most closely related to Smten. The search interface in Kaplan is tied to features specific to the Z3 solver rather than providing a generic search interface that can be used with a variety of SAT and SMT solvers. Using terminology from Rosette, Smten can be thought of as a solver-aided host language for Haskell instead of Racket. Rosette requires programs to be self-finitizing rather than providing a mechanism to support general Turing-complete computation in the description of constraints.

In Section 3.4.1 we will discuss how adding free variables to a functional language can cause problems with referential transparency. Both Kaplan and Rosette have multiple ways of instantiating free variables to cope with these problems. The problems are avoided entirely in Smten, because the interface for search in Smten preserves referential transparency.

2.8.7 Logic Programming

Logic programming languages such as Prolog [12] provide a full language solution for describing search problems, in a different style from traditional imperative and functional programming languages. A variations of Prolog that brings it closer to functional programming is Lambda Prolog [44]. Curry [24] is a functional logic programming language based on Haskell. Implementations of logic programming languages have not traditionally used SAT or SMT solvers.

2.8.8 Search in Artificial Intelligence

Schemer [65] is a language that adds support for nondeterministic computation to Lisp. Schemer allows user-level constraints for search problems to be described di-

rectly. The implementation of search in Schemer was based on a custom implementation of dependency-directed backtracking [52] rather than leveraging SAT or SMT solvers. The implementors of Schemer found the bookkeeping for search to be difficult to maintain. Smten has the benefit of leveraging SMT solvers for the search procedure and advanced techniques from Haskell for maintenance of bookkeeping.

Chapter 3

Smten Search Interface

In this chapter we present the interface for users to describe Smten search queries. The primary goals of the Smten search interface are:

- Express queries of the form $\exists x.\, \phi(x)$ where x can be any type of variable, including a variable of user-defined type, and where the constraints $\phi(x)$ can be expressed using features of a general purpose functional programming language.

- Support efficient reduction of search-space descriptions to SAT and SMT queries.

To satisfy these goals, we add a small number of primitives for describing and executing search to the Haskell programming language. Smten search queries can be described using all the features available in standard Haskell programming. We will suggest in this chapter how the specific primitives we add to Haskell support efficient reduction of Smten search descriptions to SAT and SMT, the details of which are discussed in Chapter 4. We refer to the Haskell programming language with extensions for Smten search as the Smten language.

To begin our discussion of the Smten search interface, Section 3.1 provides a brief justification of the use of Haskell as the base language for Smten, as well as a review of important features of the Haskell programming language. In Section 3.2 we present the Smten primitives for describing search spaces in terms of sets, and Section 3.3 describes the primitive for searching these spaces.

Section 3.4 presents an alternative view of Smten search as nondeterministic computation, which illustrates potential problems with preservation of referential transparency that other high-level languages for SAT and SMT face. Viewing Smten search as nondeterministic computation will also provide intuition for the behavior of search in infinitely-recursive and partially-defined search spaces, discussed in detail in Section 3.5. Section 3.6 presents a precise operational semantics of Smten search, and Section 3.7 reviews related work for describing search using Haskell.

3.1 The Haskell Language

The Smten language is based on Haskell, a general purpose, high-level functional programming language.

Haskell can be used to implement complex applications that require interaction with users through the command line, files, graphical interfaces, or other means. Using Haskell's foreign function interface, Haskell programs can call functions from other languages, such as C, and export functions to other languages. Haskell includes many programming language features supporting abstraction, encapsulation, and modularity, including higher-order functions, recursion, user-defined data types, polymorphism, operator overloading, and modules [38], which can be used to develop applications that are modular, simple, and flexible [25].

Because Haskell is a declarative and purely functional language, it is more natural to transform constraints described in Haskell to constraints supported by SAT and SMT solvers than it would be for constraints described in an imperative language.

3.1.1 Introduction to Haskell

In this section we give a brief overview of functions, user-defined data types, parametric polymorphism, ad-hoc polymorphism, monads, and the `do`-notation in Haskell.

```
1  factorial :: Integer → Integer
2  factorial n =
3    if n < 2
4       then 1
5       else n * factorial (n-1)
6
7  ackermann :: Integer → Integer → Integer
8  ackermann m n =
9    if m == 0
10      then 1
11      else if m > 0 && n == 0
12              then ackermann (m-1) 1
13              else ackermann (m-1) (ackermann m (n-1))
```

Figure 3-1: Haskell implementation of the factorial and Ackermann functions.

Functions

Figure 3-1 shows how the factorial and Ackermann functions can be defined in Haskell. Line #1 is the type signature of the factorial function, which indicates the type of factorial is a function from an Integer to an Integer, denoted `Integer -> Integer`. The type for the Ackermann function (Line #7) is denoted `Integer -> Integer -> Integer`, which indicates the Ackermann function takes two Integer arguments and produces a single Integer result.

Line #2 defines the body of the factorial function. The variable n will be bound to the input argument when the function is applied. As is typical of functional programming, these functions are defined recursively instead of using loops.

Functions in Haskell can also be defined anonymously using a lambda term. For example, the following expression describes an anonymous function that returns five more than its argument:

```
λx → x + 5
```

Anonymous functions are useful in conjunction with higher-order functions such as the `map` function from the Haskell standard library, which applies a function to every element of a list. For example, the following expression adds five to every element of

```
1 data Maybe a = Nothing | Just a
2
3 fromMaybe :: a → Maybe a → a
4 fromMaybe default x =
5   case x of
6     Just v → v
7     Nothing → default
```

Figure 3-2: Definition of `Maybe` type and `fromMaybe` function in Haskell.

the given list to produce the list `[6, 12, 9, 7]`:

```
map (λx → x + 5) [1, 7, 4, 2]
```

User-Defined Algebraic Data Types

Haskell supports polymorphic, user-defined algebraic data types. For example, Figure 3-2, on Line #1, shows a definition for the `Maybe` type in Haskell, which captures the notion of an optional value of some type `a`. The `Maybe` type has two constructors. The constructor `Nothing` indicates there is no value present, while the constructor `Just` indicates there is a value present, and that value is the argument to the constructor `Just`. For example, the expression `Just 5` is an expression of type `Maybe Integer` with value 5.

Pattern matching is used to deconstruct a value of type `Maybe` to access its fields. For example, a common function used when dealing with the `Maybe` type is the `fromMaybe` function, which returns the value of the `Maybe` if it is present, otherwise it returns a default value. The implementation of `fromMaybe` is shown on Line #3 of Figure 3-2.

The `fromMaybe` function is a polymorphic function: it can be used to deconstruct a value of type `Maybe` for any concrete type associated with the type variable `a`.

Ad-hoc Polymorphism (Type Classes)

Haskell supports ad-hoc polymorphism via type classes, where the behavior of a function depends on the specific type it is instantiated with. For example, the `Eq`

```
1  class Eq a where
2      (==) :: a → a → Bool
3
4  instance (Eq a) ⇒ Eq (Maybe a) where
5      (==) a b =
6         case a of
7           Nothing →
8              case b of
9                Nothing → True
10               Just _ → False
11          Just x →
12             case b of
13               Nothing → False
14               Just y → (x == y)
```

Figure 3-3: `Eq` class and `Maybe` instance in Haskell.

```
elem :: (Eq a) ⇒ a → [a] → Bool
elem s l =
  case l of
    [] → False
    (x:xs) → (s == x) || (elem s xs)
```

Figure 3-4: Ad-hoc polymorphic `elem` function in Haskell.

class, whose definition is shown in Figure 3-3 on Line #1, provides a way to overload the notion of equality for each type. An instance of the `Eq` for the `Maybe` type is shown on Line #4. The instance applies to any type `a` that also is an instance of the `Eq` class.

Having provided an instance of `Eq` for the `Maybe` type, objects of type `Maybe` can be used in ad-hoc polymorphic functions requiring `Eq`, such as the `elem` function shown in Figure 3-4, which returns true if a given element is a member of a list. The membership test used by the `elem` function is based on the user-defined notion of equality. The `Maybe` instance for the `Eq` class makes it possible to write code such as the following, which tests whether `Maybe 4` is an element of a given list:

```
elem (Maybe 4) [Maybe 1, Nothing, Maybe 4, Maybe 7]
```

```
1  class Monad m where
2    return :: a → m a
3    (>>=) :: m a → (a → m b) → m b
4
5  instance Monad Maybe where
6    return x = Just x
7
8    (>>=) x f =
9       case x of
10       Nothing → Nothing
11       Just v → f v
```

Figure 3-5: `Monad` class and `Maybe` instance in Haskell.

Monadic Computation

An important class in Haskell is the `Monad` class, shown on Line #1 in Figure 3-5. The `Monad` class represents the class of types that perform computation with an implicit context. The important methods of the `Monad` class are `return`, which returns a result without accessing the implicit context, and `>>=`, called bind, which describes how values and the implicit context are threaded between computations.

The `Maybe` type is an example of a `Monad` instance (Line #5). The `Maybe` monad represents computations that may fail. The `return` method creates a computation that never fails. The bind method creates a computation that evaluates the first argument to bind. If evaluating the first argument to bind fails, the overall bind computation fails. Otherwise the bind computation is the application of the second argument to the successful result of the first argument.

For example, consider a computation that looks up the value of two integers in a list, and returns their sum. Either lookup may fail. This could be expressed directly as:

```
lookup :: String → [(String, Integer)] → Maybe Integer
lookup = ...

sumFromList :: [(String, Integer)]
                    → String → String → Maybe Integer
sumFromList l a b =
  case lookup a l of
     Nothing → Nothing
     Just x →
        case lookup b l of
            Nothing → Nothing
            Just y → Just (x+y)
```

Using the `Monad` methods instead, the implementation of `sumFromList` simplifies as follows:

```
sumFromList l a b =
  lookup a l >>= λx →
  lookup b l >>= λy →
  return (x+y)
```

This way of describing monadic computations is so common, Haskell has a special syntax for it called `do`-notation. Using `do`-notation, the implementation of `sumFromList` is expressed as follows:

```
sumFromList l a b = do
  x ← lookup a l
  y ← lookup b l
  return (x+y)
```

Note that the expressions `lookup a l` and `lookup b l` have type `Maybe Integer`, while the variables `x` and `y` have type `Integer`.

Other common monads in Haskell are the list monad, which performs a compu-

tation in the context of nondeterminism, the state monad, which performs a computation in the context of some implicit state, and the IO monad, which performs computation that implicitly affects the state of the world. The IO monad is a primitive monad that allows the user to express input and output behavior in a purely functional manner in Haskell.

3.1.2 String Constraints in Haskell

Figure 3-6 shows how the substring and regular expression constraints from our string constraint solver could be implemented in Haskell. The functions defined in Figure 3-6 are suitable for use in a brute-force implementation of the string constraint solver. Aside from changing loops into recursive calls, this description is similar to the pseudocode for the same constraints given in Figure 2-3 and Figure 2-6 in Section 2.2.

In this code, regular expressions are represented using algebraic data types (Line #15). The `match` function (Line #18) uses pattern matching and higher-order functions to test whether a string matches a regular expression.

3.2 Smten Search-Space Description

In this section we describe the Smten search extension to the Haskell language for describing search spaces. Our requirement is for search-space descriptions to support all the standard features of Haskell and for search-space descriptions to compose in a modular fashion.

For SAT and SMT solvers, the search space is described with a query of the form $\exists x.\, \phi(x)$. This form of query explicitly separates the search-space description into a completely unconstrained space according to the type of variable x and constraints on the space described by $\phi(x)$. Separating the search space into an unconstrained space x and restrictions $\phi(x)$ reduces modularity, especially when the variable x is being used as a control variable, because $\phi(x)$ depends on how the unconstrained space x is described.

```
contains :: String → String → Bool
contains x s =
   if s == ""
      then x == ""
      else (prefix x s || contains x (tail s))

prefix :: String → String → Bool
prefix x s =
  case x of
    "" → True
    (x0:xs) → case s of
                  "" → False
                  (s0:ss) → x0 == s0 && prefix xs ss

data RegEx = Empty | Epsilon | Atom Char
 | Star RegEx | Concat RegEx RegEx | Or RegEx RegEx

match :: RegEx → String → Bool
match r s =
  case r of
    Empty → False
    Epsilon → s == ""
    Atom c → s == c:""
    Concat a b → any (match2 a b) (splits [0..length s] s)
    Or a b → match a s || match b s
    Star x →
      if s == ""
        then True
        else any (match2 x r) (splits [1..length s] s)

match2 :: RegEx → RegEx → (String, String) → Bool
match2 a b (sa, sb) = match a sa && match b sb

splits :: [Int] → String → [(String, String)]
splits ns x :: map (λn → splitAt n x) ns
```

Figure 3-6: Haskell code for string constraints.

```
data Space a = ...

empty  :: Space a
single :: a → Space a
union  :: Space a → Space a → Space a
map    :: (a → b) → Space a → Space b
join   :: Space (Space a) → Space a
```

Figure 3-7: Smten search-space primitives.

For the brute-force approach, instead of posing a satisfiability query, we constructed a set of candidate strings that were tested one by one. In both cases, we are describing a set of elements: either the set of values x that satisfy the formula $\phi(x)$, or the explicit set of candidate strings. The brute-force approach does not suffer from the same problems of modularity as the satisfiability query approach.

A key insight for Smten search is we can describe primitive search queries using sets of elements, as we do in the brute-force search approach, and the search procedure is responsible for decomposing the search space into free variables and constraints.

This section presents Smten's `Space` type for describing search spaces, which will be passed to the Smten search procedure. A small number of primitive operations can be used to describe rich search spaces, with the modularity and expressiveness of functional programming in Haskell.

3.2.1 The Space Type

Figure 3-7 shows the abstract data type and operations for describing search spaces in Smten. Conceptually, an expression of type `Space a` describes a set of elements of type `a`. However, it is helpful to think of `Space a` as describing a search space for elements of type `a`, because the Smten runtime does not need to construct the entire set to search for an element of it. The primitives can be used to construct, combine, and restrict sets of elements.

Figure 3-8 shows the set interpretation of the primitives for describing search spaces. The primitive `empty` is the empty search space and `single` e is a search

Space Primitive	Set Interpretation
`empty`	$\emptyset$
`single` e	$\{e\}$
`union` s_1 s_2	$s_1 \cup s_2$
`map` f s	$\{f(e) \mid e \in s\}$
`join` s_s	$\{e \mid e \in s_i, s_i \in s_s\}$

Figure 3-8: The meaning of Smten search-space primitives.

space with a single element e. The primitive `union` s_1 s_2 is the union of two search spaces. The `map` primitive applies a function to each element in the search space. The primitive `join` collapses a search for search spaces into a search for elements of those search spaces. Using a combination of the `map` and `join` primitives makes it possible to filter elements out of a search space, as we will see in the following examples.

3.2.2 Example Search Spaces

Example 3.1 {`True`, `False`}**.**

```
free_Bool :: Space Bool
free_Bool = union (single True) (single False)
```

Example 3.1 shows a description of a search space for a boolean that can have value either `True` or `False`, described using the set {`True`, `False`}. The set is formed by taking the union of a singleton set with single element `True` and another singleton set with single element `False`. Use of this space corresponds to introducing a new free boolean variable in a SAT or SMT query.

Example 3.2 {`"foo"`, `"sludge"`}**.**

```
ex3.2 :: Space String
ex3.2 = union (single "foo") (single "sludge")
```

Example 3.2 shows a description of a search space for a string that can have value either `"foo"` or `"sludge"`, described using the set {`"foo"`, `"sludge"`}. This example shows that user-level domain concepts, such as strings, can be represented directly, as easily as the primitive boolean type.

Example 3.3 {"foo", "sludge"}.

```
ex3.3 :: Space String
ex3.3 = map (λp → if p then "foo" else "sludge") free_Bool
```

Example 3.3 shows another description of the same set of elements as Example 3.2, but in terms of the search space from Example 3.1. It uses the `map` primitive to apply a function to each element of the search space `free_Bool`. The value `True` is mapped to the string `"foo"`, and `False` is mapped to the string `"sludge"`.

Example 3.3 shows that search spaces can be constructed using the composition of other search spaces. Note that this example includes both expressions of type `Space Bool` and `Space String` simultaneously. The way search spaces are composed together can influence the SAT or SMT query generated for that search space.

Example 3.4 {("hi", "foo"), ("hi", "sludge"), ("bye", "foo"), ("bye", "sludge")}.

```
ex3.4 :: Space (String, String)
ex3.4 = join (map (λa →
          join( map (λb → single (a,b))
                    (union (single "foo") (single "sludge"))))
            (union (single "hi") (single "bye")))
```

Example 3.4 shows how the `join` and `map` primitives can be used to describe a cross-product set, containing all pairs of strings where the first element is drawn from one set, and the second element is drawn from another. This describes the set {("hi", "foo"), ("hi", "sludge"), ("bye", "foo"), ("bye", "sludge")}. Broken down into steps, initially we have the set {"hi", "bye"}. The `map` primitive is used to transform each individual element of the initial set to a set of elements:

{{("hi", "foo"), ("hi", "sludge")}, {("bye", "foo"), ("bye", "sludge")}}

Finally the `join` primitive flattens the set of set into a single set of elements:

{("hi", "foo"), ("hi", "sludge"), ("bye", "foo"), ("bye", "sludge")}

If we were using set notation, we might describe this as:

$$\{(a, b) \mid a \in \{\texttt{"hi"}, \texttt{"bye"}\}, b \in \{\texttt{"foo"}, \texttt{"sludge"}\}\}$$

We will show in later examples how Haskell's special syntax for monadic computation can be used to express this cross product example with syntax much closer to the mathematical set notation shown here.

Example 3.5 {"hi"}**.**

```
ex3.5 :: Space String
ex3.5 = join (map (λa →
                  if length a == 2 then single a else empty))
             (union (single "hi") (single "bye")))


length :: [a] → Int
length l =
 case l of
   [] → 0
   (x:xs) → 1 + length xs
```

Example 3.5 shows how `map` and `join` can be combined to filter elements from a set. In this example, we filter all elements of the set {"hi", "bye"} whose length is 2. The approach is similar to the cross product example. Initially we have the set {"hi", "bye"}. The `map` primitive is used to map each element of the initial set to a singleton set with that element's value if the element has length 2, or the empty set otherwise: {{"hi"}, ∅}. Finally, the `join` primitive is used to flatten the set, removing all empty-set elements and concatenating the singleton sets: {"hi"}.

Note that the condition used to filter the set is an ordinary function in Haskell. It is not limited to constraints supported by SAT and SMT solvers. This is crucial for supporting descriptions of search spaces at the level of the user's domain instead of low level primitive types and constraints.

```
do {e}                  = e
do { v ← e ; stmts}     = join (map (λv → do {stmts}) e)
do {e; stmts}           = do {_ ← e; stmts}
do {let decls; stmts} = let decls in do {stmts}
```

Figure 3-9: Haskell `do`-notation for the Space type.

Example 3.6 $sx \cap sy$**.**

```
intersect :: (Eq a) ⇒ Space a → Space a → Space a
intersect sx sy = join (map (λx →
          join( map (λy → if x == y then single x else empty)
              sy) sx))
```

Example 3.6 shows how a function that takes the intersection of two sets can be implemented using the `map` and `join` primitives, using a combination of cross product and filtering from the previous examples. The equality operator is user defined.

3.2.3 Taking Advantage of Haskell's `do`-Notation

The `Space` type and primitives presented in Figure 3-7 form a *monad* [43]. This means we can provide an instance of the `Monad` type class for `Space` and use Haskell's special syntactic support for monadic computation: the `do`-notation. When used with the `Space` type, the `do`-notation serves as a generalization of set comprehensions.

Figure 3-9 shows how the `do`-notation for the `Space` type desugars into calls of `map` and `join`.

Example 3.7.

```
ex3.7 :: Space String
ex3.7 = do
  a ← union (single "hi") (single "bye")
  b ← union (single "foo") (single "sludge")
  single (a, b)
```

Example 3.7 shows the cross product example using `do`-notation. This shows how `do`-notation is like set compressions. It can be read as the set of pairs (a, b) such that a is drawn from the set {`"hi"`, `"bye"`} and b is drawn from the set {`"foo"`, `"sludge"`}.

Example 3.8.

```
ex3.8 :: Space String
ex3.8 = do
  p ← free_Bool
  single (if p then "foo" else "sludge")
```

Example 3.9.

```
ex3.9 :: Space String
ex3.9 = do
  a ← union (single "hi") (single "bye")
  if length a == 2
    then single a
    else empty
```

Example 3.8 shows how `do`-notation can be used to rewrite Example 3.3. Example 3.9 shows how `do`-notation can be used to rewrite the filter in Example 3.5. We can also factor out the conditional expression into a separate function, `guard`, to provide filtering in more of a set-comprehension style:

Example 3.10.

```
guard :: Bool → Space ()
guard p = if p then single () else empty


ex3.10 :: Space String
ex3.10 = do
  a ← union (single "hi") (single "bye")
  guard (length a == 2)
  single a
```

```
1  strs_length :: Int → Space String
2  strs_length n =
3    if n == 0
4       then single ""
5       else do
6          c ← choose ['a'..'z']
7          str ← strs_length (n-1)
8          single (c:str)
9
10 choose :: [a] → Space a
11 choose l =
12   case l of
13     [] → empty
14     x:xs → union (single x ) (choose xs)
```

Figure 3-10: Smten search space for strings of a given length.

3.2.4 Search Spaces for the String Constraint Solver

In this section we show examples of more realistic Smten search-space descriptions using the string constraint solver example.

Figure 3-10 shows the description of a search space for all strings of a given length n. If the length is 0, the `strs_length` function (Line #1) returns the singleton set with the empty string `""` (Line #4). Otherwise `strs_length` uses `do`-notation to create the cross product of all possible choices for the first character with all possible choices for a string of smaller length (Line #5). The `choose` function (Line #10) is a helper function that converts a list of choices into a Smten search space.

Figure 3-11 shows a search-space description for the space of strings matching a regular expression. The description mirrors the semantics of regular expression described in Figure 2-5 from Section 2.2. One important point about this search-space description is it uses unbounded recursion in the recursive call on Line #14. We will come back to the question of how this should be interpreted when executing search in Section 3.5.

These descriptions of search spaces for strings can be combined with more traditional constraints using the `guard` function.

```
1  strs_regex :: RegEx → Space String
2  strs_regex r =
3    case r of
4      Empty → empty
5      Epsilon → single ""
6      Atom c → single [c]
7      Concat a b → do
8        sa ← strs_regex a
9        sb ← strs_regex b
10       single (sa ++ sb)
11     Or a b → union (strs_regex a) (strs_regex b)
12     Star x → union (single "") (do
13       sx ← strs_regex x
14       sr ← strs_regex r
15       single (sx ++ sr)
16      )
```

Figure 3-11: Smten search space for strings from a regular language.

Example 3.11.

```
ex3.11 :: RegEx → [String] → Space String
ex3.11 r xs = do
  str ← strs_regex r
  guard (all (λx → contains x str) xs)
  single str
```

Example 3.11 shows the description of a search space for strings based on the template of a regular expression r that satisfy all of the substring constraints xs. The `contains` predicate was defined in Figure 3-6 to work on ordinary Haskell strings. Here it is being reused in the description of a search space, conceptually being applied to a set of strings.

This example corresponds to the versions of the string constraint solver implementation that enumerates the possible strings by regular expression instead of length. Unlike the pseudocode we presented for satisfiability-based approach for that implementation, there is no explicit notion of control variables. The `contains` function does not have to be modified, or aware of the control variables, and the specific de-

```
data Solver = yices2 | z3 | miniSat | ...

search :: Solver → Space a → IO (Maybe a)
```

Figure 3-12: The Smten `search` primitive

cisions taken to construct the string from the regular expression are not exposed to the top-level search.

3.3 Executing Smten Search

In addition to describing a search space, we need to extend Haskell with some way to execute a search based on a SAT or SMT solver. For SAT and SMT, at the high level we provided primitive Sat and Witness functions to call the solver and retrieve the satisfying assignment. Section 2.6.2 discussed how the interface to the SAT or SMT solver can be much more complicated in practice. For Smten, we provide the primitive `search`, which combines Sat and Witness into a single function to search for an element of a Smten search space.

Figure 3-12 shows the type of `search` primitive for executing search in Haskell. We have specifically chosen the type and interpretation of `search` to be amenable to implementation by SAT and SMT solver.

The meaning of the `search` primitive, given a search space corresponding to a set of expressions s, is:

$$\texttt{search}\ \_\ s = \begin{cases} \texttt{return Nothing} & \text{if } s = \emptyset \\ \texttt{return (Just } e\texttt{)} & \text{for some } e \in s \end{cases}$$

The following points are noteworthy about the `search` primitive:

- The solver to use for performing search is specified as an explicit argument, independent of the search space being searched. This makes it very easy to experiment with different SAT and SMT solvers. For example, switching be-

tween using the Yices2 and Z3 solvers requires modification of only a single line of code. This is in contrast to hand-coded implementations of SAT and SMT-based tools, where the specific solver being used is often hard-coded and tied up with the description of the search space.

- The `search` primitive searches for a single element of the search space. This corresponds to SAT and SMT solvers, which primarily search for a single satisfying assignment or model for the given formula and constraints. Because the only way to inspect a search space is by accessing a single element, the entire search space never has to be constructed. This restriction is appropriate for the combinatorial search problems we are interested in solving, because for the most part, they are interested in finding a single solution or determining there is no solution. If more than one solution to a search space is desired, that can be achieved using repeated calls to `search`, filtering elements out of the search space that were returned by previous invocations of `search`.

- If the search space is not empty, a witness is returned as an expression with the same type as the elements of the search space, rather than as a model or assignment to primitive variables in the SAT or SMT query. As a consequence, the user does not have to know how the search space is constructed to use the result of `search`. There is no extra procedure required to interpret the witness, even when the search is for elements of a user-defined data type.

- The `search` primitive is nondeterministic. If there are multiple elements in the search space, any one of those results may be returned. The `search` primitive belongs to the `IO` monad because it is nondeterministic, which is conventional in Haskell for impure functions.

- A convenient side effect of `search` belonging to the `IO` monad is that searches cannot be nested. Consequently, the Smten search procedure does not have to solve queries about queries, which are not directly supported by SAT and SMT solvers. Instead, a user interested in solving queries about queries can

Example 3.12.

```
ex3.12 :: Space String
ex3.12 = do
  a ← union (single "hi") (single "bye")
  b ← union (single "foo") (single "sludge")
  let c = a ++ b
  guard (length c /= 6)
  single c

main :: IO ()
main = do
  result ← search yices2 ex3.12
  case result of
    Nothing → putStrLn "No Solution"
    Just s → putStrLn ("Solution: " ++ s)
```

manually decompose their search in a domain specific way into multiple calls to the `search` primitive.

Example 3.12 shows how the `search` primitive can be used to search a search space. In this example, the yices2 solver is used to search for a string with length other than six that comes from the concatenation of a string drawn from the set {`"hi"`, `"bye"`} with a string drawn from the set {`"foo"`, `"sludge"`}. This program is nondeterministic, it could output any of the following:

- `"Solution: hifoo"`
- `"Solution: hisludge"`
- `"Solution: byesludge"`

Example 3.13 shows how the search primitive can be used for the string constraint solver. For this example we describe the search space of all strings of a given length, restricted to those matching the regular expression r and substring constraints xs. We start by calling `search` for the smallest possible length, and increase the length one-by-one until either a string is found, or the upper bound on the length has been exceeded.

Example 3.13.

```
strsolve :: Int → Int → RegEx → [String] → IO ()
strsolve l h r xs = do
  if l > h
     then putStrLn "NoSuchString"
     else do
       result ← search yices2 (do
          str ← strs_length l
          guard (match r str)
          guard (all (λx → contains x str) xs)
          single str
        )
       case result of
         Maybe s → putStrLn ("Found String: " ++ s)
         Nothing → strsolve (l+1) h r xs
```

At this point, except for showing how to parse a string constraint specification into the variables l, h, r, and xs, we have provided a complete implementation of the string constraint solver using Smten search. The implementation is simple, expressed at the user level, and leverages features for abstraction, modularity, and encapsulation from Haskell. It would be easy to add additional constraints, reorganize the search for strings by regular expression instead of by length, or experiment with a different SMT solver backend.

3.4 Smten Search as Nondeterminism with Backtracking

In Section 3.5 we will discuss how search should be treated for search spaces that are infinite or not completely defined. It will be helpful for that discussion to think of Smten search in terms of nondeterminism with backtracking instead of in terms of sets. We have already seen a glimpse of this interpretation with the use of control variables in satisfiability queries (Section 2.4). These control variables represent nondeterministic choices. The SAT or SMT solver serves as an oracle that chooses those

decisions that lead to a successful result.

More precisely, nondeterministic computation has some way of expressing choice. Sometimes this is with a primitive called `amb`, in reference to the ambiguous functions of McCarthy [40]. We will simply called it `choice`. The `choice` operator takes two arguments, one of which is nondeterministically chosen to be the result of the operator.

Nondeterministic computation with *backtracking* means a computation can succeed with a result or fail, expressed using primitives `success`, which takes the result as an argument, and `fail`, which takes no arguments. It is the responsibility of the system to ensure the choice taken in the `choice` operator leads to a succeeding computation if possible. In practice this is implemented by greedily picking one of the alternatives, finishing the computation, backtracking to the `choice` operator and using the other alternative if the computation failed.

Nondeterminism with backtracking is a general form of search problem that can be used to concisely describe the combinatorial search applications we are interested in [65].

Example 3.14.

```
str_regex :: RegEx → String
str_regex r =
  case r of
    Empty → fail
    Epsilon → success ""
    Atom c → success (c:"")
    Concat a b → success (str_regex a ++ str_regex b)
    Or a b → choice (str_regex a) (str_regex b)
    Star x → choice (single "")
                    (str_regex x ++ str_regex r)
```

Example 3.14 shows a procedure using the syntax of Haskell that nondeterministically computes a string from the language of a given regular expression, assuming primitives `choice`, `success`, and `fail` have been provided.

Note the similarity between this and the implementation of `strs_regex` presented in Figure 3-11 from Section 3.2 using the Smten search-space primitives to describe the set of all strings belonging to the language of the regular expression. The `choice`, `success`, and `fail` primitives correspond to the Smten search-space primitives `union`, `single`, and `empty` respectively.

Using sets is a better way to describe nondeterministic computation than providing a `choice` primitive, because it resolves problems with preserving referential transparency in a functional programming language.

3.4.1 Preserving Referential Transparency

One challenge that arises when extending a functional language with support for nondeterministic computation is how to preserve referential transparency. For example, consider the following code, which assumes the existence of a `choice` primitive for introducing nondeterministic choice:

```
let x = choice 1 2
in x + x
```

The variable x represents a nondeterministic choice between the values 1 and 2. The overall expression could have two different interpretations. A natural interpretation would be to say it is twice whatever the choice is for x. In that case, it represents a nondeterministic choice between the values 2 and 4. If referential transparency is preserved, however, this expression should be equivalent to the following:

```
choice 1 2 + choice 1 2
```

In this case, there are 3 reasonable possibilities: 2, 3, and 4, not just 2 and 4.

Many attempts at integrating SAT and SMT solvers in a fundamental way into a functional language based on nondeterminism leave the interpretation of this ambiguous, or use different approaches to partially resolve the ambiguity. For example, in Rosette [56] there are two different ways to declare a variable, one form makes the nondeterministic choice immediately, the other delays the nondeterministic choice for every use. The Sketch language for program synthesis faces the same problem. They

choose to distinguish between normal functions, where the nondeterministic choice is made first, with its result duplicated if the function is called multiple times, and generators, where the nondeterministic choice is inlined before the choice is made. There is still ambiguity in Sketch, however, because it is not always clear when inlining of generators occurs. For instance, are generators inlined before loops are unrolled, in which case the same choice is used for every iteration of the loop, or are generators inlined after loops are unrolled, in which case a different decision can be taken for each iteration of the loop?

Hughes and O'Donnell [26] studied the issue of preserving referential transparency when introducing nondeterministic computation in a function programming language and found using sets to describe the nondeterministic computation is a satisfactory approach.

Using the above example, we distinguish between variables that have a set type, representing a nondeterministic choice yet to be made, and variables that refer to elements of the set, which represent a choice already made.

The following two examples demonstrate how both interpretations can be expressed explicitly using sets to describe nondeterministic computation:

```
let x = {1, 2}
in {a + a | a in x}
```

```
let x = {1, 2}
in {a + b | a in x, b in x}
```

In the first case, it is clear a single value from the set of choices will be added to itself, resulting in either 2 or 4. In the second case, it is clear variables a and b may be different, and the possible results are 2, 3, or 4. Note that even though sets are used to describe the nondeterministic choice, in the approach of Hughes and O'Donnell, the entire set is not computed, because the user is interested in only a single result. The use of sets is entirely descriptive, not prescriptive of how to implement the nondeterminism. Nondeterminism as described by Hughes and O'Donnell can be

implemented efficiently because they don't have support for backtracking, so decisions can be made greedily. Smten adds full support for for backtracking. In this case use of sets is still appropriate, but the implementation is more complicated.

The key takeaway is that Smten search-space descriptions can be thought of as nondeterministic computation, where conceptually a SAT or SMT solver is used as an oracle for the nondeterministic choices. This interpretation will be useful in understanding the desired behavior of Smten search when search spaces are infinite or not completely defined.

3.5 Turing-Complete Search Computation

Haskell is a Turing-complete programming language. It is possible to write programs that use unbounded recursion or otherwise fail to terminate. This is an important feature for implementing complex applications solving problems based on dynamic inputs from users and for which there may not necessarily exist decision procedures, as is the case for many of the combinatorial search problems we are interested in using Smten to solve.

Because we have done nothing to explicitly disallow it, it is possible to describe search spaces involving infinite or nonterminating computations in Smten. We have already seen one example of this in the implementation of `strs_regex` in Section 3.2.4. As we will show, the facility for describing infinite or nonterminating computations in search is useful for working with infinite search spaces and data types, such as the infinite length strings, sequences of states, and programs that arise naturally in string constraint solving, model checking, and program synthesis. This raises an important question, however, because the problem of SAT is entirely finite, and SMT solvers have only limited supported for handling queries involving infinite data types such as unbounded integers or real numbers. What behavior should we expect when searching a search space described with infinite or nonterminating computation?

We will use the following guiding intuition to understand the behavior of search in the presence of infinite or nonterminating computation:

- To show an expression e belongs to a search space s, the system needs to construct only enough of the search space s to verify it contains e.
- To show a search space s is empty, the system must construct the entire space.

To get a better sense of the different ways nonterminating computation can appear when describing a search space and the behavior we would expect when searching that space, we will consider a handful of examples. In these examples, we will use the symbol $\perp$ to represent an infinitely recursive or nonterminating computation. We think of $\perp$ as representing an expression that has not yet been fully evaluated, but in time may eventually resolve to a value.

3.5.1 Search for Partially-Defined Elements

Example 3.15 $\{\perp\} \cup \{\texttt{"foo"}\}$**.**

```
ex3.15 :: Space String
ex3.15 = union (single ⊥) (single "foo")
```

Example 3.15 shows a search space that contains a partially-defined element. Even if we do not know what value $\perp$ will eventually resolve to, if it resolves at all, we know some interesting things about this set:

- The set is not empty.
- The set contains the totally defined element `"foo"`.
- The set contains an element $\perp$ whose value is determined by the result of a long-running or infinite computation.

Note that the element $\perp$ may eventually resolve to `"foo"`, in which case this describes a set with a single element. Otherwise this describes a set with exactly two elements. Distinguishing between these two cases is unnecessary because the `search` primitive searches for only a single result.

According to our guiding intuition, search should be allowed to return `Just "foo"`, because clearly the search space contains the element `"foo"`. Search can also return

`Just` $\perp$, because clearly the search space contains the element $\perp$, even if we do not know what $\perp$ is yet. In Haskell, because evaluation is nonstrict, a thunk for the element represented with $\perp$ can be returned before the value is fully evaluated. Regardless of what value is returned, search should not fail to terminate.

The user can test whether the search space is empty or not by checking if the result of search has the constructor `Nothing` or `Just`. If the result returned is `Just "foo"`, the user can inspect the value `"foo"` and perform additional computation on it. If the result returned is `Just` $\perp$, as soon as the user tries to look at the value of $\perp$, it will force the computation of $\perp$, which may fail to terminate.

One way to interpret the behavior of search for search spaces with partially-defined elements is to say search is nonstrict in the value of elements of the search spaces. Because `search` does not care what the value it returns is, only that it returns some value in the search space, there is no reason for it not to be able to return a partially-defined value.

3.5.2 Search in Partially-Defined Spaces

Example 3.16 $\perp \cup \{$`"foo"`$\}$**.**

```
ex3.16 :: Space String
ex3.16 = union ⊥ (single "foo")
```

Example 3.16 shows a search space that is partially defined. The first argument to `union` is $\perp$, representing a completely undefined set. Contrast this with Example 3.15, where the first argument to `union` was $\{\perp\}$, a set with a single element.

Like Example 3.15, the search space described in Example 3.16 clearly contains the element `"foo"`, so `Just "foo"` is an acceptable result of searching the space. Unlike Example 3.15, the search space might not contain any other element; the set represented by $\perp$ could eventually resolve to the empty set. Because the search space clearly contains the element `"foo"`, we would prefer search in the space returns eventually, even if the set represented by $\perp$ cannot be computed.

There are important performance consequences for allowing `Just "foo"` to be re-

turned when searching this search space. Taking the nondeterministic view of search, infinite paths of the computation can be pruned if not needed (in this case, paths of computation depending on the value of the first argument to `union`). It is also the case that *long* paths can be pruned. The search procedure can return those elements of the space that are easiest to find, without having to construct complicated parts of the search space, and without having to determine whether the search space could be fully constructed.

Example 3.17 $\bot \cup \emptyset$**.**

```
ex3.17 :: Space String
ex3.17 = union ⊥ empty
```

Example 3.17 shows a partially defined search space using `empty` for the second argument to `union` instead of `single "foo"`. For this space, search will fail to terminate if $\bot$ fails to resolve, because the search space could either be empty or contain some element depending on how $\bot$ resolves.

3.5.3 Infinite Search Spaces

It is possible to use infinite recursion in Haskell to describe an infinite search space. Infinite search spaces occur frequently at the user-level for combinatorial search applications. For example, model checkers search over the space of all possible counter-examples, and this space can be infinite if there is no bound on the length of counter-examples. Similarly, for program synthesis, the space of possible programs is infinite unless explicitly and unnaturally bounded in size.

The following example demonstrates a `Space` expression using infinite recursion in Haskell to describe the infinite set of strings formed by repeated concatenation of the character `'a'`:

Example 3.18 $\{\epsilon, \texttt{"a"}, \texttt{"aa"}, \ldots\}$**.**

```
strA :: Space String
strA = union (single "") (map (λs → 'a':s) strA)
```

In Example 3.18, the search space `strA` is defined recursively as the union of a space containing the string `""`, and the space of all strings in `strA` with the character `'a'` prepended. Any string in the set described by `strA` can be returned by `search`, which follows from our previous discussion if we partially unroll the definition of `strA`, replacing the recursive call to `strA` with $\perp$.

For example, replacing the recursive call to `strA` immediately with $\perp$ produces the space:

```
strA = union (single "") ⊥
```

The empty string clearly belongs to this space, so search can return `Just ""`.

If the recursive call to `strA` is unrolled once before replacing it with $\perp$, the search space simplifies to:

```
strA = union (single "") (union (single "a") ⊥)
```

The string `"a"` clearly belongs to this space.

By partially unrolling the space further and further, any string of characters `'a'`, such as the string `"aaaaaaa"`, clearly belongs to the search space and can be returned by `search`.

Infinite search space can be used in meaningful ways in the construction of other, potentially finite, search spaces. For example, we can ask for a string in `strA` whose length is between 2 and 4:

Example 3.19 {`"aa"`, `"aaa"`, `"aaaa"`}**.**

```
strAlen2 :: Space String
strAlen2 = do
  s ← strA
  guard (length s ≥ 2 && length s ≤ 4)
  return s
```

This describes the finite set {`"aa"`, `"aaa"`, `"aaaa"`}. Ideally this set is semantically equivalent to one constructed without the use of infinite recursion:

Example 3.20.

```
strAlen2' :: Space String
strAlen2' = union (single "aa")
            (union (single "aaa")
                   (single "aaaa"))
```

It is important to allow finite sets to be described in terms of infinite sets, because manually constructing the finite set, as in Example 3.20, may not be possible without violating modularity.

Unfortunately, in general it is not always possible to treat `strAlen2` and `strAlen2'` as semantically equivalent search spaces. In particular, when filtering elements out of the search space, the only way to determine if the set is empty or not is to know if every element has been filtered, which requires testing every element of the set. We will see why this is so when looking at search spaces with infinite data types, such as unbounded integers. For now, we simply present an example that demonstrates the problem:

Example 3.21.

```
search slv (do
  s ← strA
  guard (elem 'b' s)
  return s
 )
```

It is infeasible in general to expect the system to determine that none of the strings described by `strA` contain the character `'b'`.

3.5.4 Implicitly Infinite Search Spaces

Some SMT solvers provide basic support for queries involving unbounded integers. In particular, there are decision procedures for queries involving unbounded integers

restricted to the operations of linear integer arithmetic. Queries such as these can be described in Smten:

Example 3.22.

```
ex3.22 :: Space Integer
ex3.22 = do
  x ← free_Integer
  guard (factorial x == 100)
  return x
```

In this example, `free_Integer` is meant to describe the space of all integers. We would like queries involving free integers to be supported in Smten, especially if there are obvious solutions, such as in the following example:

Example 3.23.

```
ex3.23 :: Space Integer
ex3.23 = do
  x ← free_Integer
  guard (factorial x == 120)
  return x
```

In general, it is not possible to solve all search queries involving unbounded integers, because it is an undecidable problem [39].

In order to understand the behavior of queries involving unbounded integers, we treat them as any other infinite search space. The `free_Integer` space can be described directly in Smten as an infinite search space, and we use this description to interpret the semantics of queries involving unbounded integers:

```
free_Integer :: Space Integer
free_Integer =
  let free_Nat = union (single 0) (map (+ 1) free_Nat)
  in union free_Nat (map negate free_Nat)
```

3.5.5 Conditional Partially-Defined Search Spaces

Note that it is possible for search spaces to be partially defined conditioned on an element of another search space. Consider the following examples:

Example 3.24.

```
ex3.24 :: Space String
ex3.24 = do
  x ← free_Bool
  if x
    then ⊥
    else single x
```

Example 3.25.

```
ex3.25 :: Space String
ex3.25 = do
  x ← free_Bool
  if x
    then ⊥
    else empty
```

Example 3.24 and Example 3.25 are both partially defined search spaces, conditioned on the element x from the search space `free_Bool`. In Example 3.24, if the value of x is `False`, then the search space is a singleton set containing the element `False`, otherwise the search space is undefined. `Just False` should be a valid result of searching this space, because there is a path in the search space that leads to the element `False` that does not depend on the undefined part of the search space.

In Example 3.25, if the value of x is `False`, then the search space is empty, otherwise the search space is undefined. Search in this space should fail to terminate, because the undefined part of the search space must be considered to determine if the search space is empty.

These two examples show that there is a difference in behavior we can expect if the search space is empty or not.

3.5.6 Unreachably Infinite Search Spaces

Example 3.26.

```
1 ex3.26 :: Space Bool
2 ex3.26 = do
3   x ← free_Bool
4   if x && not x
5     then ⊥
6     else single x
```

Example 3.26 shows an example of an unreachably infinite search space. For every choice of x, the condition on Line #4 evaluates to `False`, resulting in the search space `single` x. In other words, the undefined part of the search space on Line #5 is unreachable. Search in this space should return `Nothing`. In general, if there are no reachable infinite paths, search should terminate in a bounded number of steps. This example is important, because when we rely on a SAT or SMT solver to evaluate the condition, it is often not obvious the undefined part of the search space is unreachable. We will revisit this example in Section 4.3 when we discuss how Smten search spaces are reduced to SAT and SMT queries.

In the next section we formalize the semantics of Smten search to clarify the corner cases when Turning-complete computation is involved in search-space descriptions.

3.6 Precise Semantics for Smten

In this section we provide a precise semantics for search in Smten. This is to clarify the behavior of search, especially with respect to nonterminating computation in search-space descriptions. We will also use these semantics as the foundation of our presentation of the syntax directed description of the Smten implementation in

$$
\begin{array}{llll}
 & x & \in & \textit{Variable} \\
\text{Type} & T & ::= & T_1 \to T_2 \mid \texttt{Unit} \mid T_1\texttt{*}T_2 \mid T_1\texttt{+}T_2 \mid \texttt{IO}\ T \mid \texttt{S}\ T \\
\text{Term} & e, f, s & ::= & x \mid \lambda x_T\ .\ e \mid f\ e \\
 & & \mid & \texttt{unit} \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
 & & \mid & \texttt{inl}_T\ e \mid \texttt{inr}_T\ e \mid \texttt{case}\ e\ f_1\ f_2 \mid \texttt{fix}\ f \\
 & & \mid & \texttt{return}_{io}\ e \mid e\ \texttt{>>=}_{io}\ f \mid \texttt{search}\ s \\
 & & \mid & \texttt{empty}_T \mid \texttt{single}\ e \mid \texttt{union}\ s_1\ s_2 \mid \texttt{map}\ f\ s \mid \texttt{join}\ s \\
\text{Abbr} & \multicolumn{3}{l}{\texttt{Maybe}\ T = \texttt{Unit + }T, \quad \texttt{Just}\ e = \texttt{inr}\ e, \quad \texttt{Nothing} = \texttt{inl unit}} \\
 & \multicolumn{3}{l}{\texttt{err}_T = \texttt{fix}\ (\lambda x_T\ .\ x)}
\end{array}
$$

Figure 3-13: Syntax of KerSmten types and terms.

Section 4.2. Unlike other work on formalizing the semantics of languages for high-level use of SAT and SMT solvers (most notably Kaplan [34]), the semantics we provide are given entirely in terms of the Smten search primitives, and do not fundamentally refer to or rely on behavior of a SAT or SMT solver. The semantics presented here have been formalized and mechanically checked using the Coq proof assistant [29].

To focus on the search aspects of Smten and simplify the presentation, we give semantics for a reduced kernel language, called KerSmten, instead of the full Smten language based on Haskell. KerSmten is a nonstrict, strongly typed lambda calculus with pairs, disjoint sums, general recursion through `fix`, input/output (IO) computations in the Haskell-monadic style, and the Smten search interface. In the semantics, the `search` primitive does not take a solver as an argument, because the solver argument to `search` in Smten affects only the performance of search and not the semantics of search.

Figure 3-13 shows the syntax of types and terms for KerSmten. The type $T_1 \to T_2$ is the type of a function that takes an argument of type T_1 and returns a result of type T_2. The `Unit` type is a type with one inhabitant. The type T_1`*`T_2 denotes the product of types T_1 and T_2. The type T_1`+`T_2 denotes the disjoint sum of types T_1 and T_2. IO computations have type `IO` T, where T is the type of the result of the IO computation. Finally, search-space computations have type `Space` T, abbreviated as `S` T, where T is the type of expression being searched for.

We will sometimes use the abbreviation `Maybe` T, corresponding to Haskell's `Maybe` type, for the disjoint sum of types `Unit` and T, representing an expression that evaluates to either a value of type T or the explicit absence of any value.

For the syntax of terms in Figure 3-13, the variables e, f, and s all denote expressions, with the intent that e is used for general expressions, f is used for expressions describing functions, and s is used for expressions describing search spaces.

KerSmten is an explicitly typed monomorphic language. Lambda terms are annotated with the type of the input variable: `inl` and `inr` are both annotated with the unused part of the disjoint sum and `empty` is annotated with the result type for that search-space computation. This information, along with the structure of a term, can be used to uniquely determine the type of a KerSmten term. Formally, the notation $\Gamma \vdash e : T$ is the assertion that term e is well-typed in the environment Γ, which maps variable names to types.

Figure 3-14 lists typing judgements that assign types to each well-typed term. The typing judgments for pairs, disjoint sums, and general recursion through `fix` are standard, the IO computations are typed as in Haskell, and the search primitives have types corresponding to the Haskell types presented for them in Section 3.2 and Section 3.3. To reduce clutter, we sometimes omit explicit type annotations when not relevant.

We organize the description of the KerSmten semantics into four parts. The first part describes the semantics of pure evaluation, where no IO or search-space computations are performed. Next we discuss search-space computation, which is split into search-space expansion and search-space reduction, and finally we discuss IO computation.

3.6.1 Pure Evaluation

Pure evaluation reduces a KerSmten expression to a pure value without performing any IO or search-space computations. The primitives for IO and search computations are considered values with respect to pure evaluation. Figure 3-15 shows the subset of terms that are the values of pure evaluation and gives a small-step structured

t-var $\quad \Gamma[x \mapsto T] \vdash x : T$

t-abs $\quad \dfrac{\Gamma[x \mapsto T_1] \vdash e : T_2}{\Gamma \vdash \lambda x_{T_1}.e : T_1 \to T_2}$

t-app $\quad \dfrac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$

t-unit $\quad \Gamma \vdash \texttt{unit} : \texttt{Unit}$

t-pair $\quad \dfrac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 \texttt{*} T_2}$

t-fst $\quad \dfrac{\Gamma \vdash e : T_1 \texttt{*} T_2}{\Gamma \vdash \texttt{fst}\ e : T_1}$

t-snd $\quad \dfrac{\Gamma \vdash e : T_1 \texttt{*} T_2}{\Gamma \vdash \texttt{snd}\ e : T_2}$

t-inl $\quad \dfrac{\Gamma \vdash e : T_1}{\Gamma \vdash \texttt{inl}_{T_2}\ e : T_1 \texttt{+} T_2}$

t-inr $\quad \dfrac{\Gamma \vdash e : T_2}{\Gamma \vdash \texttt{inr}_{T_1}\ e : T_1 \texttt{+} T_2}$

t-fix $\quad \dfrac{\Gamma \vdash f : T \to T}{\Gamma \vdash \texttt{fix}\ f : T}$

t-case $\quad \dfrac{\Gamma \vdash e : T_1 \texttt{+} T_2 \qquad \Gamma \vdash f_1 : T_1 \to T_3 \qquad \Gamma \vdash f_2 : T_2 \to T_3}{\Gamma \vdash \texttt{case}\ e\ f_1\ f_2 : T_3}$

t-return $\quad \dfrac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{return}_{io}\ e : \texttt{IO}\ T}$

t-bind $\quad \dfrac{\Gamma \vdash e_1 : \texttt{IO}\ T_1 \qquad \Gamma \vdash e_2 : T_1 \to \texttt{IO}\ T_2}{\Gamma \vdash e_1\ \texttt{>>=}_{io}\ e_2 : \texttt{IO}\ T_2}$

t-search $\quad \dfrac{\Gamma \vdash s : \texttt{S}\ T}{\Gamma \vdash \texttt{search}\ s : \texttt{IO}\ (\texttt{Maybe}\ T)}$

t-empty $\quad \Gamma \vdash \texttt{empty}_T : \texttt{S}\ T$

t-single $\quad \dfrac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{single}\ e : \texttt{S}\ T}$

t-union $\quad \dfrac{\Gamma \vdash s_1 : \texttt{S}\ T \qquad \Gamma \vdash s_2 : \texttt{S}\ T}{\Gamma \vdash \texttt{union}\ s_1\ s_2 : \texttt{S}\ T}$

t-map $\quad \dfrac{\Gamma \vdash f : T_1 \to T_2 \qquad \Gamma \vdash s : \texttt{S}\ T_1}{\Gamma \vdash \texttt{map}\ f\ s : \texttt{S}\ T_2}$

t-join $\quad \dfrac{\Gamma \vdash s : \texttt{S}\ (\texttt{S}\ T)}{\Gamma \vdash \texttt{join}\ s : \texttt{S}\ T}$

Figure 3-14: KerSmten typing rules.

Pure Evaluation

Pure Value $v ::= \lambda x_T \,.\, e \mid \texttt{unit} \mid (e_1, e_2) \mid \texttt{inl}_T\ e \mid \texttt{inr}_T\ e$
$\mid \texttt{return}_{io}\ e \mid e_1\ \texttt{>>=}_{io}\ e_2 \mid \texttt{search}\ e$
$\mid \texttt{empty}_T \mid \texttt{single}\ e \mid \texttt{union}\ s_1\ s_2 \mid \texttt{map}\ f\ s \mid \texttt{join}\ s$

st-beta	$(\lambda x \,.\, e_1)\ e_2$	$\rightarrow_e$	$e_1.[e_2/x.]$
st-fst	$\texttt{fst}\ (e_1, e_2)$	$\rightarrow_e$	e_1
st-snd	$\texttt{snd}\ (e_1, e_2)$	$\rightarrow_e$	e_2
st-inl	$\texttt{case}\ (\texttt{inl}\ e)\ f_1\ f_2$	$\rightarrow_e$	$f_1\ e$
st-inr	$\texttt{case}\ (\texttt{inr}\ e)\ f_1\ f_2$	$\rightarrow_e$	$f_2\ e$
st-fix	$\texttt{fix}\ f$	$\rightarrow_e$	$f\ (\texttt{fix}\ f)$

st-fst-a $\dfrac{e \rightarrow_e e'}{\texttt{fst}\ e \rightarrow_e \texttt{fst}\ e'}$ *st-snd-a* $\dfrac{e \rightarrow_e e'}{\texttt{snd}\ e \rightarrow_e \texttt{snd}\ e'}$

st-app-a $\dfrac{f \rightarrow_e f'}{f\ e \rightarrow_e f'\ e}$

st-case-a $\dfrac{e \rightarrow_e e'}{\texttt{case}\ e\ f_1\ f_2 \rightarrow_e \texttt{case}\ e'\ f_1\ f_2}$

Figure 3-15: Smten rules for pure evaluation

operational semantics for pure evaluation in KerSmten. The transition $e_1 \rightarrow_e e_2$ represents a single step of pure evaluation. The small steps for pure evaluation are standard for call-by-name evaluation.

Though we do not show the proofs here, pure evaluation is type-sound and deterministic.

3.6.2 Search-Space Computation

Search-space computation is performed on terms of type `S` T. In addition to pure evaluation, all occurrences of `union`, `map`, and `join` are eliminated, resulting in either the empty search space `empty` or a search space with a single value `single` e.

Search-space computation is split into two parts. The first part, described using small steps with the notation $s_1 \rightarrow_{s\uparrow} s_2$, eliminates top level occurrences of `map` and `join`, resulting in an expanded set of elements described using `empty`, `single`, and `union`. We call this phase search-space expansion.

The second part of search computation is described using small steps with the notation $s_1 \rightarrow_{s\downarrow} s_2$. This phase, which we call search-space reduction, eliminates occurrences of `union`, selecting an arbitrary result.

Search-Space Expansion

Figure 3-16 shows the subset of terms that are the values of search-space expansion and the small step semantics of search-space expansion. As with pure evaluation, search-space expansion is type-sound and deterministic.

Search-Space Reduction

Figure 3-16 shows the values and small steps for search-space reduction. Note that search-space reduction includes search-space expansion.

Search-space reduction is type sound, but is not deterministic: either the left or the right argument to `union` can be selected.

Search-Space Expansion

S1 Value v_{s1} ::= $\texttt{empty}_T$ | $\texttt{single}\ e$ | $\texttt{union}\ s_1\ s_2$

sts-map-empty	$\texttt{map}\ f\ \texttt{empty}$	$\rightarrow_{s\uparrow}$	$\texttt{empty}$
sts-map-single	$\texttt{map}\ f\ (\texttt{single}\ e)$	$\rightarrow_{s\uparrow}$	$\texttt{single}\ (f\ e)$
sts-map-union	$\texttt{map}\ f(\texttt{union}\ s_1\ s_2)$	$\rightarrow_{s\uparrow}$	$\texttt{union}\ (\texttt{map}\ f\ s_1)\ (\texttt{map}\ f\ s_2)$
sts-join-empty	$\texttt{join}\ \texttt{empty}$	$\rightarrow_{s\uparrow}$	$\texttt{empty}$
sts-join-single	$\texttt{join}\ (\texttt{single}\ s)$	$\rightarrow_{s\uparrow}$	s
sts-join-union	$\texttt{join}\ (\texttt{union}\ s_1\ s_2)$	$\rightarrow_{s\uparrow}$	$\texttt{union}\ (\texttt{join}\ s_1)\ (\texttt{join}\ s_2)$

sts-pure $\dfrac{s \rightarrow_e s'}{s \rightarrow_{s\uparrow} s'}$ *sts-map-a* $\dfrac{s \rightarrow_{s\uparrow} s'}{\texttt{map}\ f\ s \rightarrow_{s\uparrow} \texttt{map}\ f\ s'}$

sts-join-a $\dfrac{s \rightarrow_{s\uparrow} s'}{\texttt{join}\ s \rightarrow_{s\uparrow} \texttt{join}\ s'}$

Search-Space Reduction

S Value v_s ::= $\texttt{empty}_T$ | $\texttt{single}\ e$

sts-union-left	$\texttt{union}\ (\texttt{single}\ e)\ s$	$\rightarrow_{s\downarrow}$	$\texttt{single}\ e$
sts-union-right	$\texttt{union}\ s\ (\texttt{single}\ e)$	$\rightarrow_{s\downarrow}$	$\texttt{single}\ e$
sts-union-not-right	$\texttt{union}\ s\ \texttt{empty}$	$\rightarrow_{s\downarrow}$	s
sts-union-not-left	$\texttt{union}\ \texttt{empty}\ s$	$\rightarrow_{s\downarrow}$	s

sts-union-a2 $\dfrac{s_2 \rightarrow_{s\downarrow} s_2'}{\texttt{union}\ s_1\ s_2 \rightarrow_{s\downarrow} \texttt{union}\ s_1\ s_2'}$

sts-union-a1 $\dfrac{s_1 \rightarrow_{s\downarrow} s_1'}{\texttt{union}\ s_1\ s_2 \rightarrow_{s\downarrow} \texttt{union}\ s_1'\ s_2}$ *sts-s1* $\dfrac{s \rightarrow_{s\uparrow} s'}{s \rightarrow_{s\downarrow} s'}$

Figure 3-16: Smten rules for search-space expansion and reduction

IO Computation

IO Value v_{io} ::= $\texttt{return}_{io}$

stio-bind-return $(\texttt{return}_{io}\ e_1)\ \texttt{>>=}_{io}\ e_2 \rightarrow_{io} e_2\ e_1$

stio-search-empty $\texttt{search empty} \rightarrow_{io} \texttt{return}_{io}\ \texttt{Nothing}$

stio-search-single $\texttt{search}\ (\texttt{single}\ e) \rightarrow_{io} \texttt{return}_{io}\ (\texttt{Just}\ e)$

stio-bind-a $$\frac{e_1 \rightarrow_{io} e_1'}{e_1\ \texttt{>>=}_{io}\ e_2 \rightarrow_{io} e_1'\ \texttt{>>=}_{io}\ e_2}$$

stio-pure $$\frac{e \rightarrow_e e'}{e \rightarrow_{io} e'}$$

stio-search-a $$\frac{s \rightarrow_{s\downarrow} s'}{\texttt{search}\ s \rightarrow_{io} \texttt{search}\ s'}$$

Figure 3-17: Smten rules for IO computation

3.6.3 IO Computation

IO computation applies to terms of type `IO` T and is where search computations are executed. Though we have not included them here to avoid distraction, additional IO primitives could be added for performing input and output, which is why we call this IO computation. Note that IO computations cannot be run from within a search computation.

Figure 3-17 shows the values and small steps for IO computation. The notation $e_1 \rightarrow_{io} e_2$ is used for IO reductions. IO computation is type sound.

3.6.4 Examples Revisited

Here we apply the operational semantics to explain possible behaviors of search using examples from the previous section.

Example 3.27 $\{\bot\} \cup \{\texttt{"foo"}\}$**.**

```
ex3.27 :: Space String
ex3.27 = union (single ⊥) (single "foo")
```

There is no search-space expansion to perform when searching the space from Example 3.27. Search-space reduction can reduce this to either `single` $\bot$ by *sts-*

union-left or `single "foo"` by *sts-union-right*. As a consequence, after applying *stio-search-single*, the result is either $\texttt{return}_{io}$ (`Just` $\bot$) or $\texttt{return}_{io}$ (`Just "foo"`). This demonstrates that partially defined elements of the search space may be returned by `search`.

Example 3.28.

```
ex3.28 :: Space String
ex3.28 = do
  x ← free_Bool
  if x
    then ⊥
    else single x
```

With de-sugaring, Example 3.28 becomes:

```
join (map (λx → if x then ⊥ else single x)
          union (single True) (single False))
```

After application of *sts-map-union*:

```
join (union (map (λx → if x then ⊥ else single x) (single True))
            (map (λx → if x then ⊥ else single x) (single False)))
```

Then *sts-join-union*:

```
union (join (map (λx → if x then ⊥ else single x) (single True)))
      (join (map (λx → if x then ⊥ else single x) (single False)))
```

Then *sts-union-a2* with *sts-map-single*:

```
union (join (map (λx → if x then ⊥ else single x) (single True)))
      (join (single False))
```

Then *sts-union-a2* with *sts-join-single*:

```
union (join (map (λx → if x then ⊥ else single x) (single True)))
      (single False)
```

Then *sts-union-right*:

```
single False
```

Which allows the return of `Just False` by *stio-search-single*.

If instead we used the example where the else branch is `empty` instead of a singleton set, we would have reached:

```
union (join (map (λx → if x then ⊥ else empty) (single True)))
      empty
```

By *sts-union-not-right* this becomes:

```
join (map (λx → if x then ⊥ else empty) (single True))
```

Applying *sts-map-single* results in:

```
join ⊥
```

Which will fail to terminate, applying repeatedly the rule for *sts-join-a*.

3.7 Related Work

Various approaches have been used to add search features to functional programming languages and Haskell.

Hughes and O'Donnell demonstrated that sets can be used to describe nondeterministic computation in a functional language while preserving referential transparency [26]. Their work did not include support for backtracking, and as a consequence, they could implement search by greedily picking a single element at each choice point. The combinatorial search applications we are solving with Smten fundamentally rely on the ability to do backtracking search.

Moggi showed how monads can be used to add different notions of computation to lambda calculus [43], and Wadler showed how to employ these monads to simplify functional programming [61, 62]. Smten's `Space` type is very similar to the list monad for describing nondeterministic computations, but the interface to `Space` is more restricted: it searches for only a single element, disallows nested search, and does not impose an ordering on the search the way the list monad does.

Multiple approaches have been suggested to provide better functionality and performance than the list monad. Work by Kiselyov et al. [32] and monadic constraint programming [46] provide more flexibility and control over how the search is performed. Neither of these approaches are targeted towards using SAT and SMT to improve the performance of the search.

Chapter 4

Smten Search Approach

In this chapter we describe how SAT and SMT solvers are used to implement Smten search. Section 4.1 walks through an example from the string constraint solver to illustrate the idea behind how Smten search is performed. Section 4.2 presents the search approach formally. In Section 4.3 we present a mechanism for supporting Turing-complete computation in search-space descriptions, and Section 4.4 discusses optimizations that improve the performance of the Smten search approach in practice.

4.1 Intuition for Smten Search

In this section we walk through the Smten search procedure for a specific example, to build intuition for the Smten search approach in general. The example is the following specific instance of string constraint solving:

```
matches ab(c|d)(dd|ee); contains de;
```

Imagine we are using the string constraint solver that organizes the search by enumerating the strings from the language of the regular expression, then restricting that space to those strings satisfying the length and substring constraints. This specific string constraint specification could be encoded directly as the following Smten

search space:

```
do str ← strs_regex /ab(c|d)(dd|ee)/
   if contains "de" str
     then single str
     else empty
```
(4.1)

For purposes of illustration, we have described the regular expression using regular expression syntax between forward slashes on the first line. In practice the regular expression would be expressed directly using the `RegEx` data type, or parsed into the `RegEx` data type from a string.

The `do`-notation in (4.1) will be desugared into primitive calls of `map` and `join`:

```
join (map (λstr → if contains "de" str
                    then single str
                    else empty)
          (strs_regex /ab(c|d)(dd|ee)/))
```
(4.2)

Brute-Force Search The brute-force approach to searching (4.2) is to enumerate all strings belonging to the language of the regular expression:

$$\{\texttt{"abcdd"}, \texttt{"abcee"}, \texttt{"abddd"}, \texttt{"abdee"}\}$$

Next, the `contains` function is applied to each string in turn, removing those strings from the set that do not contain `"de"`, leaving us with the only solution: `"abdee"`.

Smten Search In Smten, instead of enumerating the entire set of strings, we exploit similarities in the strings to give a compact representation of the set of strings.

The compact representation of a set of strings is a single string with SAT variables embedded to indicate different possible values of the string:

$$\left\{ \texttt{"ab} \begin{bmatrix} v_1?\texttt{c} \\ \texttt{:d} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \texttt{"} \right\}$$

The notation $\begin{bmatrix} v_1?\texttt{c} \\ \texttt{:d} \end{bmatrix}$ represents a single character whose value depends on a boolean variable. The value is `'c'` when v_1 is `true`, and `'d'` when v_1 is `false`.

We use two SAT variables in the compact representation of the set of strings: v_1 represents the choice between characters `'c'` and `'d'` for the third character of the string, and v_2 represents the choice between the sequence `"dd"` and `"ee"` for the suffix of the string. Using this representation for the set of strings in place of the call to `regex_strs` in (4.2) gives:

$$\begin{aligned} &\texttt{join (map (}\lambda\texttt{str} \rightarrow \texttt{ if contains "de" str} \\ &\qquad\qquad\qquad\qquad\qquad\texttt{then single str} \\ &\qquad\qquad\qquad\qquad\qquad\texttt{else empty)} \\ &\qquad\left\{ \texttt{"ab} \begin{bmatrix} v_1?\texttt{c} \\ \texttt{:d} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \texttt{"} \right\}\texttt{)} \end{aligned} \tag{4.3}$$

We can now evaluate constraints, such as the `length` and `contains` constraints, on the compact representation of the set of strings. For example, the `length` function can be applied to the compact representation and can compute that the length of the string is five, even without knowing the values of the individual characters:

$$\texttt{map length} \left\{ \texttt{"ab} \begin{bmatrix} v_1?\texttt{c} \\ \texttt{:d} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \texttt{"} \right\} = \{5\}$$

Because of our compact representation, the `length` function had to be applied only once to determine that the length of all four strings described by the compact

representation is five. The `contains` function, unlike the `length` function, depends on the values of the characters in the string. We can still apply the `contains` function to the compact representation of the set of strings, however. In this case, the result of `contains` will be a compact representation of a set of booleans expressed in terms of the SAT variables in the compact representation of the set of strings:

$$\texttt{map (contains "de")} \left\{ \texttt{"ab} \begin{bmatrix} v_1?\texttt{c} \\ \texttt{:d} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \texttt{"} \right\}$$
$$= \left\{ \begin{bmatrix} (\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2) & ? & \texttt{True} \\ & : & \texttt{False} \end{bmatrix} \right\}$$

This is more efficient than evaluating `contains` on each individual string, because it shares the fact that `"de"` does not appear at the beginning of any of the strings.

The application of `map` in (4.3) involves a function slightly more complicated than the `contains` constraint. The function body has a conditional expression used to produce a search space based on the result of `contains "de" str`. The result of applying the function is a compact representation of a search space:

$$\texttt{join} \left\{ \begin{bmatrix} (\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2) & ? & \texttt{single "ab} \begin{bmatrix} v_1?\texttt{c} \\ \texttt{:d} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \texttt{"} \\ & : & \texttt{empty} \end{bmatrix} \right\} \tag{4.4}$$

The `join` primitive can be applied to what is conceptually a set containing a single set, resulting in a final, compact representation of the search-space description:

$$\begin{bmatrix} (\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2) & ? & \texttt{single "ab} \begin{bmatrix} v_1?\texttt{c} \\ \texttt{:d} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \begin{bmatrix} v_2?\texttt{d} \\ \texttt{:e} \end{bmatrix} \texttt{"} \\ & : & \texttt{empty} \end{bmatrix} \tag{4.5}$$

Represented in this form, it is clear the search space contains an element exactly when the formula $(\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2)$ is satisfied. We can pose a query to a SAT

solver to check if the formula is satisfiable, and if so, we can combine the satisfying assignment μ with the string argument to `single` to form a concrete member of the search space:

$$\texttt{"ab}\begin{bmatrix} v_1\texttt{?c} \\ \texttt{:d} \end{bmatrix}\begin{bmatrix} v_2\texttt{?d} \\ \texttt{:e} \end{bmatrix}\begin{bmatrix} v_2\texttt{?d} \\ \texttt{:e} \end{bmatrix}\texttt{"} \text{ with } \mu = \{v_1 = \texttt{false}, v_2 = \texttt{false}\}$$

gives `"abdee"`.

In summary, the approach to Smten search is to represent sets of elements compactly by embedding SAT or SMT variables into normal expressions. It is more efficient to evaluate constraints on the compact representation than evaluating the constraints on each individual element of the set being represented. A search-space description is reduced to a compact representation of a search space of the form:

$$\begin{bmatrix} \phi \texttt{ ? single } e \\ \texttt{: empty} \end{bmatrix}$$

The search space contains an element exactly when the formula ϕ is satisfiable, and the satisfying assignment from the SAT solver can be used to compute a concrete element of the search space.

4.2 Syntax-Directed Approach to Smten Search

In this section we present a more precise, syntax-directed approach for constructing a SAT formula from a Smten description of a search problem. The Smten approach to search is more effective than a brute-force approach because it avoids constructing the entire set of elements in a search space, it avoids applying constraints to every element of the set, and it relies on SAT and SMT solvers for efficient backtracking search.

A key insight for producing SAT formulas from Smten search-space descriptions is to introduce two new forms of expressions to the Smten language. The first new

$$
\begin{array}{rcl}
e, f, s & ::= & x \mid \lambda x_T \,.\, e \mid f\ e \\
& \mid & \texttt{unit} \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
& \mid & \texttt{inl}_T\ e \mid \texttt{inr}_T\ e \mid \texttt{case}\ e\ f_1\ f_2 \mid \texttt{fix}\ f \\
& \mid & \texttt{return}_{io}\ e \mid e\ \texttt{>>=}_{io}\ f \mid \texttt{search}\ s \\
& \mid & \texttt{empty}_T \mid \texttt{single}\ e \mid \texttt{union}\ s_1\ s_2 \mid \texttt{map}\ f\ s \mid \texttt{join}\ s \\
& \mid & \phi\ ?\ e_1\ :\ e_2 \mid \{e \mid \phi\} \\
\\
\phi & ::= & \texttt{true} \mid \texttt{false} \mid v \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\
& \mid & \texttt{ite}\ \phi_1\ \phi_2\ \phi_3
\end{array}
$$

Figure 4-1: Augmented syntax of KerSmten terms.

$$
\textit{t-phi}\quad \frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash (\phi\ ?\ e_1\ :\ e_2) : T} \qquad \textit{t-set}\quad \frac{\Gamma \vdash e : T}{\Gamma \vdash \{e \mid \phi\} : \mathtt{S}\ T}
$$

Figure 4-2: Typing judgements for ϕ-conditional and set expressions.

form of expression is a ϕ-conditional expression that parameterizes an expression by a boolean formula, allowing arbitrary expressions to be dependent on the assignment of SAT variables. The second is a set expression representing a set of expressions by a combination of a boolean formula and an assignment-parameterized expression. The augmented syntax of KerSmten with the ϕ-conditional expression and set expression is shown in Figure 4-1, and typing rules for the new forms of expression are given in Figure 4-2. The syntax of boolean formulas ϕ is the same as in Figure 2-17, with the addition of `ite` ϕ_1 ϕ_2 ϕ_3, which is syntactic sugar for $(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \phi_3)$.

ϕ-Conditional Expression (ϕ ? e_1 : e_2) The ϕ-conditional expression is a conditional expression parameterized by the value of the boolean formula ϕ. The value of the expression is e_1 for all boolean assignments under which ϕ evaluates to `true`, and e_2 for all assignments under which ϕ evaluates to `false`.

The expression ϕ ? e_1 : e_2 is well-typed with type T if both e_1 and e_2 are well-typed with type T.

An example of a ϕ-conditional expression is the expression $(v \wedge w)$? 1 : 2, which is an expression with value 1 for any assignment where both boolean variables v and w are `true`, and 2 otherwise. Sometimes we prefer to use the notation from the

previous section, which emphasizes the alternative choices graphically:

$$\begin{bmatrix} v \wedge w \ ?\ 1 \\ \qquad\quad : 2 \end{bmatrix}$$

We call an expression containing ϕ-conditional sub-expressions *partially concrete* in contrast to fully concrete expressions.

We use the notation $e[\mu]$ to refer to the concrete value of a partially concrete expression e under given boolean assignment μ, where all ϕ-conditional sub-expressions have been eliminated. For example, if we have that $\mu_1 = \{(v, \texttt{true}), (w, \texttt{true})\}$ and $\mu_2 = \{(v, \texttt{false}), (w, \texttt{true})\}$, then we have that $(v \wedge w \ ?\ 1\ :\ 2)[\mu_1] = 1$ and $(v \wedge w \ ?\ 1\ :\ 2)[\mu_2] = 2$.

Set Expression $\{e \mid \phi\}$ The set expression is a canonical form for expressions of type `Space` a:

$$\{e \mid \phi\} = \begin{bmatrix} \phi\ ?\ \texttt{single}\ e \\ \quad : \texttt{empty} \end{bmatrix}$$

In this form, each satisfying assignment μ of the boolean formula ϕ corresponds to a different element, $e[\mu]$, of the search space. The search space is empty exactly when ϕ is unsatisfiable. The set of expressions represented by set expression $\{e \mid \phi\}$ is the set of possible values of e for satisfying assignments of the boolean formula ϕ:

$$\{e[\mu] \mid \phi[\mu] = \texttt{true}\}$$

For example, the set expression $\{(v \wedge w)\ ?\ 1\ :\ 2 \mid v \vee w\}$ represents the set $\{1, 2\}$, because both μ_1 and μ_2 from above are satisfying assignments of the formula $v \vee w$. In contrast, the set expression $\{(v \wedge w)\ ?\ 1\ :\ 2 \mid v \wedge w\}$, with conjunction in the formula instead of disjunction, represents the singleton set $\{1\}$, because μ_1 is the only satisfying assignment of the formula $v \wedge w$.

The type of a set expression is `S` T, where T is the type of expression e. We refer to e as the body of the set expression $\{e \mid \phi\}$.

$$
\begin{array}{lrcl}
\text{Pure Value} & v & ::= & \lambda x_T \,.\, e \mid \texttt{unit} \mid (e_1, e_2) \mid \texttt{inl}_T\ e \mid \texttt{inr}_T\ e \\
& & \mid & \texttt{return}_{io}\ e \mid e_1\ \texttt{>>=}_{io}\ e_2 \mid \texttt{search}\ e \\
& & \mid & \texttt{empty}_T \mid \texttt{single}\ e \mid \texttt{union}\ s_1\ s_2 \mid \texttt{map}\ f\ s \mid \texttt{join}\ s \\
& & \mid & \phi\ ?\ e_1\ :\ e_2 \mid \{e \mid \phi\}
\end{array}
$$

$$
\begin{array}{lrcl}
\textit{st-beta-phi} & (\phi\ ?\ f_1\ :\ f_2)\ e & \rightarrow_e & \phi\ ?\ (f_1\ e)\ :\ (f_2\ e) \\
\textit{st-fst-phi} & \texttt{fst}\ (\phi\ ?\ e_1\ :\ e_2) & \rightarrow_e & \phi\ ?\ (\texttt{fst}\ e_1)\ :\ (\texttt{fst}\ e_2) \\
\textit{st-snd-phi} & \texttt{snd}\ (\phi\ ?\ e_1\ :\ e_2) & \rightarrow_e & \phi\ ?\ (\texttt{snd}\ e_1)\ :\ (\texttt{snd}\ e_2) \\
\textit{st-case-phi} & \texttt{case}\ (\phi\ ?\ e_1\ :\ e_2)\ f_1\ f_2 & \rightarrow_e & \phi\ ?\ (\texttt{case}\ e_1\ f_1\ f_2)\ :\ (\texttt{case}\ e_2\ f_1\ f_2)
\end{array}
$$

Figure 4-3: KerSmten pure evaluation with ϕ-conditional expressions.

Each of the primitives `empty`, `single`, `union`, `map`, and `join` are evaluated at runtime to construct a set expression representing the appropriate set of elements. The primitive (`search` s) is implemented using a SAT solver as follows:

1. Evaluate the expression s, which results in the construction of a set expression:

$$
\begin{bmatrix}
\phi\ ?\ \texttt{single}\ e \\
\quad :\ \texttt{empty}
\end{bmatrix}
$$

2. Run the SAT solver on the formula ϕ. If the result is `Unsat`, then the set s is empty and we return `Nothing`. Otherwise the solver gives us an assignment μ, and we return the result `Just` $e[\mu]$, because $e[\mu]$ belongs to the set s.

To evaluate a search space to a set expression, we must augment pure evaluation to work in the presence of ϕ-conditional and set expressions, and we must define a new evaluation strategy for search-space computations.

Augmenting Pure Evaluation Figure 4-3 shows the augmented values and rules for KerSmten pure evaluation. Both the ϕ-conditional expression and set expression are considered values with respect to pure evaluation. For the set expression, this is consistent with the rest of the search-space primitives. For the ϕ-conditional expression, because it can be of any type, this introduces a new kind of value for every type. Consequently, we need to augment pure evaluation with rules to handle this new kind of value.

The effect of rules *st-beta-phi*, *st-fst-phi*, *st-snd-phi*, and *st-case-phi* is to push primitive operations inside of the ϕ-conditional expression. The duplication of primitive operations is not as severe as the duplication of function calls that occurs when using the brute-force approach because functions whose control flow is independent of their arguments can be executed once, instead of once for every argument.

For example, the expression:

$$(\lambda\ x\ .\ (x, x))\ \left[\begin{array}{r} \phi\ ?\ e_1 \\ :\ e_2 \end{array}\right]$$

reduces with standard beta substitution (*st-beta*) to:

$$(\ \left[\begin{array}{r} \phi\ ?\ e_1 \\ :\ e_2 \end{array}\right],\ \left[\begin{array}{r} \phi\ ?\ e_1 \\ :\ e_2 \end{array}\right]\ )$$

The brute-force approach would be to re-evaluate this function for e_1 and e_2 separately. If e_1 and e_2 are themselves partially concrete, the function would need to be evaluated for each of the possibly exponential number of concrete arguments using the brute-force approach, but just once using our approach.

We discuss in Section 4.4.1 how we can canonicalize partially concrete expressions to avoid duplication in the primitive operations as well.

Search-Space Evaluation Figure 4-4 gives a new set of rules for evaluating search-space expressions to set expressions. Note that these rules indicate the search primitives are strict, which is not what we desire ultimately. In this section we focus on what set expressions are constructed from search-space descriptions, and later we discuss how the runtime should generate these expressions to properly handle Turing-complete computation in search-space descriptions.

To understand why these rules produce correct set expressions, it is often helpful to view set expressions as the canonical form of partially concrete search spaces instead of as a set of expressions.

$$
\begin{array}{lrcll}
\textit{sx-empty} & \texttt{empty} & \to_{sx} & \{\bot \mid \texttt{false}\} & \\
\textit{sx-single} & \texttt{single}\ e & \to_{sx} & \{e \mid \texttt{true}\} & \\
\textit{sx-union} & \texttt{union}\ \{e_1 \mid \phi_1\}\ \{e_2 \mid \phi_2\} & \to_{sx} & \{v\ ?\ e_1\ :\ e_2 \mid \texttt{ite}\ v\ \phi_1\ \phi_2\}, & v \text{ fresh} \\
\textit{sx-map} & \texttt{map}\ f\ \{e \mid \phi\} & \to_{sx} & \{f\ e \mid \phi\} & \\
\textit{sx-join} & \texttt{join}\ \{s \mid \phi\} & \to_{sx} & \phi\ ?\ s\ :\ \texttt{empty} & \\
\textit{sx-phi} & \phi\ ?\ \{e_1 \mid \phi_1\}\ :\ \{e_2 \mid \phi_2\} & \to_{sx} & \{\phi\ ?\ e_1\ :\ e_2 \mid \texttt{ite}\ \phi\ \phi_1\ \phi_2\} &
\end{array}
$$

$$
\textit{sx-union-a1}\quad \frac{s_1 \to_{sx} s_1'}{\texttt{union}\ s_1\ s_2 \to_{sx} \texttt{union}\ s_1'\ s_2}
$$

$$
\textit{sx-union-a2}\quad \frac{s_2 \to_{sx} s_2'}{\texttt{union}\ s_1\ s_2 \to_{sx} \texttt{union}\ s_1\ s_2'}
$$

$$
\textit{sx-phi-a1}\quad \frac{s_1 \to_{sx} s_1'}{\phi\ ?\ s_1\ :\ s_2 \to_{sx} \phi\ ?\ s_1'\ :\ s_2}
$$

$$
\textit{sx-phi-a2}\quad \frac{s_2 \to_{sx} s_2'}{\phi\ ?\ s_1\ :\ s_2 \to_{sx} \phi\ ?\ s_1\ :\ s_2'}
$$

$$
\textit{sx-map-a}\quad \frac{s \to_{sx} s'}{\texttt{map}\ f\ s \to_{sx} \texttt{map}\ f\ s'} \qquad \textit{sx-join-a}\quad \frac{s \to_{sx} s'}{\texttt{join}\ s \to_{sx} \texttt{join}\ s'}
$$

$$
\textit{sx-pure}\quad \frac{s \to_{e} s'}{s \to_{sx} s'}
$$

Figure 4-4: SAT-based search-space evaluation.

sx-empty The primitive `empty` reduces to $\{\bot \mid \texttt{false}\}$ by the rule *sx-empty*. The boolean formula `false` has no satisfying assignments, so $\{\bot[\mu] \mid \texttt{false}[\mu] = \texttt{true}\}$ represents the empty set.

Interpreting the set expression as the canonical form of a partially concrete search-space expression gives:

$$\{\bot \mid \texttt{false}\} = \left\lceil \begin{array}{l} \texttt{false} \ ? \ \texttt{single} \ \bot \\ \qquad\quad : \texttt{empty} \end{array} \right\rceil$$

We use $\bot$ for the body of the set expression, but any value with the proper type could be used instead, because the body is unreachable. (We leverage this fact for an important optimization discussed in Section 4.4.2).

sx-single The primitive `single` e reduces to $\{e \mid \texttt{true}\}$ by the rule *sx-single*. The boolean formula `true` is trivially satisfiable. If e is a concrete expression, this represents a singleton set, because $e[\mu]$ is the same for all μ.

As with `empty`, treating the set expression as a canonical form makes sense:

$$\{e \mid \texttt{true}\} = \left\lceil \begin{array}{l} \texttt{true} \ ? \ \texttt{single} \ e \\ \qquad\quad : \texttt{empty} \end{array} \right\rceil$$

Note that the argument e to `single` does not need to be evaluated to reduce the expression `single` e to set expression form. The expression e may describe a non-terminating computation, but that has no effect on whether e may be returned as a result of a search.

sx-union The expression (`union` $\{e_1 \mid \phi_1\}$ $\{e_2 \mid \phi_2\}$) reduces to the set expression $\{v \ ? \ e_1 \ : \ e_2 \mid \texttt{ite} \ v \ \phi_1 \ \phi_2\}$ by *sx-union*, where v is a fresh boolean variable.

The variable v represents the choice of which argument of the union to use: an assignment of $v =$ `true` corresponds to choosing an element from the first set, and $v =$ `false` corresponds to choosing an element from the second set. The formula `ite` $v \ \phi_1 \ \phi_2$ is satisfied by satisfying assignments of ϕ_1 with $v =$ `true` and satisfying

assignments of ϕ_2 with $v = \texttt{false}$.

For example, consider the following `Space` expression representing the set $\{1, 5\}$:

```
union (single 1) (single 5)
```

This evaluates to the set expression:

$$\{v \ ? \ 1 \ : \ 5 \mid \texttt{ite} \ v \ \texttt{true} \ \texttt{true}\}$$

If this were the top-level search space, the SAT solver would be free to assign the variable v to either `true` or `false`, selecting between values 1 and 5 respectively.

In contrast, consider the following `Space` expression:

```
union (single 1) empty
```

This evaluates to:

$$\{(v \ ? \ 1 \ : \ \bot) \mid \texttt{ite} \ v \ \texttt{true} \ \texttt{false}\}$$

The only satisfying assignment of `ite` v `true false` is $\{v = \texttt{true}\}$, so the value 1 must be selected.

The `sx-union` rule is the primary means of introducing partially concrete expressions during evaluation.

sx-map The expression `map` f $\{e \mid \phi\}$ reduces to $\{f \ e \mid \phi\}$ by *sx-map*. The map reduction applies the function f to the body of the set expression, leaving the formula unchanged.

The `map` primitive can lead to arbitrary application of functions to partially concrete expressions in the body of a set expression.

sx-join The expression `join` $\{s \mid \phi\}$ reduces by the rule *sx-join* to ϕ ? s : `empty`. This reduction is easy to understand interpreting the set expression $\{s \mid \phi\}$ as the canonical form ϕ ? `single` s : `empty`. Applying the `join` operator to both branches of the ϕ-conditional expression leads to:

$$\phi \ ? \ \texttt{join} \ (\texttt{single} \ s) \ : \ \texttt{join} \ \texttt{empty}$$

Both branches trivially reduce, resulting in:

$$\phi \ ? \ s \ : \ \texttt{empty}$$

This expression must be reduced further to reach canonical form.

sx-phi A ϕ-conditional `Space` expression of the form $\phi \ ? \ \{e_1 \mid \phi_1\} \ : \ \{e_2 \mid \phi_2\}$ is reduced to the set expression $\{\phi \ ? \ e_1 \ : \ e_2 \mid \texttt{ite} \ \phi \ \phi_1 \ \phi_2\}$ by *sx-phi*. As with `join`, using intuition about the meaning of a partially concrete set expression helps to understand why this is correct if it is not immediately clear.

The term $\phi \ ? \ \{e_1 \mid \phi_1\} \ : \ \{e_2 \mid \phi_2\}$ is equivalent to the following term:

$$\begin{bmatrix} \phi \ ? \ (\phi_1 \ ? \ \texttt{single} \ e_1 \ : \ \texttt{empty}) \\ \quad : \ (\phi_2 \ ? \ \texttt{single} \ e_2 \ : \ \texttt{empty}) \end{bmatrix}$$

This can be compressed to the following form:

$$\begin{bmatrix} \texttt{ite} \ \phi \ \phi_1 \ \phi_2 \ ? \ \texttt{single} \ (\phi \ ? \ e_1 \ : \ e_2) \\ \qquad\qquad\qquad\qquad : \ \texttt{empty} \end{bmatrix}$$

This is equivalent to $\{\phi \ ? \ e_1 \ : \ e_2 \mid \texttt{ite} \ \phi \ \phi_1 \ \phi_2\}$.

4.3 Turing-Complete Search Computation

The reduction rules *sx-phi-a1*, *sx-phi-a2* are used to evaluate both branches of a ϕ-conditional set expression to set expressions before the rule *sx-phi* can be applied. These rules are overly strict, which can result in nontermination when search should terminate. It is somewhat surprising that reduction of a search-space description to a set expression may fail to terminate, even for search spaces that are completely finite from the user's point of view. Recall the following search space from an example in Section 3.5:

```
search (do
  x ← free_Bool
  if x && not x
    then ⊥
    else single x
 )
```

This describes a search for a boolean value x that is either `True` or `False`. In either case, the condition for the `if` is `False`, so the search should always follow the `else` branch. The `then` branch, which here is an infinite recursion, is unreachable.

This search is entirely finite. A brute-force approach could produce the exhaustive list of both solutions: `True` and `False`. In our implementation, this evaluates to an expression such as:

$$\left[\begin{array}{l}(x \wedge \neg x)\ ?\ (\texttt{fix}\ (\lambda y.y)) \\ \qquad\qquad : (\texttt{single}\ (x\ ?\ \texttt{True}\ :\ \texttt{False}))\end{array}\right]$$

Because we rely on the SAT solver to evaluate the condition in this expression, the implementation does not look to see that in every case the condition is `false`, so it evaluates both branches of the condition, one of which will never reduce to a set expression.

It is not unrealistic to expect that we could determine the first branch is unreachable with some simplifications to the condition in this example. In general, though, determining whether a branch is unreachable is as hard as determining whether a boolean formula is satisfiable.

One simple approach to avoid evaluating set expressions on unreachable paths is to call the SAT solver for every condition to determine whether a branch is reachable or not. Drawbacks of this approach are that the SAT solver is called for every condition, which we expect to be prohibitively expensive, and it doesn't help support general Turing-complete computation in search spaces.

We take an alternative approach that partially unrolls search spaces, using an abstraction-refinement procedure to search for elements of the search space that do not require the long running computations be completed. This approach leads to many fewer calls to a SAT solver and fully supports Turing-complete computation in search-space descriptions.

Before we describe the abstraction-refinement procedure, we will recast the problem from handling nontermination in the construction of set expressions to handling nontermination in construction of SAT formulas.

4.3.1 Lazy Evaluation of Set Expressions

In our actual implementation of Smten, we evaluate set expressions lazily, contrary to the rules presented in Figure 4-4. Because the set expressions need to be evaluated only when the formula part of the expression is used in a SAT formula, we can consider the problem of handling nontermination during construction of set expressions in terms of nontermination in construction of SAT formulas.

To understand how lazy evaluation is used, consider the example from before:

$$\left[\begin{array}{l}(x \wedge \neg x)\ ?\ (\texttt{fix}\ (\lambda y.y)) \\ \qquad\qquad : (\texttt{single}\ (x\ ?\ \texttt{True}\ :\ \texttt{False}))\end{array}\right]$$

Our goal is to reduce this expression to a set expression. Using the rule *sx-phi-a2* with *sx-single*, we can reduce the `false` branch of the conditional expression to a set expression:

$$\left[\begin{array}{l}(x \wedge \neg x)\ ?\ (\texttt{fix}\ (\lambda y.y)) \\ \qquad\qquad : \{\texttt{true} \mid (x\ ?\ \texttt{True}\ :\ \texttt{False})\}\end{array}\right]$$

We could use *sx-phi-a1* with *st-fix* to reduce the `true` branch of the conditional expression, but that does not result in any forward progress, because the expression (`fix` $(\lambda y.y)$) reduces to itself. Instead, we can construct a set expression for the `true` branch of the conditional expression by introducing *thunks* for the body and condition

$$\begin{array}{lll}
\textit{sp-rho} & \rho(\{e \mid \phi\}) \rightarrow_\phi \phi & \\
\textit{st-epsilon} & \epsilon(\{e \mid \phi\}) \rightarrow_e e & \\
\textit{sp-rho-a} & \dfrac{s \rightarrow_{s\downarrow} s'}{\rho(s) \rightarrow_\phi \rho(s')} & \textit{st-epsilon-a} \quad \dfrac{s \rightarrow_{s\downarrow} s'}{\epsilon(s) \rightarrow_e \epsilon(s')}
\end{array}$$

Figure 4-5: Rules for lazy evaluation of set expressions.

of the set expression:

$$\left[\begin{array}{l}(x \wedge \neg x)\ ?\ \{\epsilon(\texttt{fix}\ (\lambda y.y)) \mid \rho(\texttt{fix}\ (\lambda y.y))\} \\ \qquad\qquad : \{(x\ ?\ \texttt{True}\ :\ \texttt{False}) \mid \texttt{true}\}\end{array}\right]$$

Here the expression $\rho(\texttt{fix}\ (\lambda y.y))$ represents the thunk for the condition of the set expression, and $\epsilon(\texttt{fix}\ (\lambda y.y))$ represents the thunk for the body of the set expression. Evaluation of these expressions can be deferred to when either the condition or body of the set expression is needed.

Figure 4-5 shows a more precise interpretation of lazy evaluation of set expressions. A thunk is evaluated by reducing its argument to a set expression, then extracting the appropriate field. We introduce the arrow $\rightarrow_\phi$ for the ρ thunk, because it produces a formula and not a normal expression.

Using lazy evaluation, we can complete reduction of our example to a set expression using the rule *sx-phi*. The result is:

$$\{(x \wedge \neg x)\ ?\ \epsilon(\texttt{fix}\ (\lambda y.y))\ :\ (x\ ?\ \texttt{True}\ :\ \texttt{False}) \mid \texttt{ite}\ (x \wedge \neg x)\ \rho(\texttt{fix}\ (\lambda y.y))\ \texttt{true}\}$$

To search this space, we would like to call a SAT solver for the formula:

$$\texttt{ite}\ (x \wedge \neg x)\ \rho(\texttt{fix}\ (\lambda y.y))\ \texttt{true}$$

However, because of lazy evaluation, the nonterminating part of the set expression reduction has been embedded as a nonterminating part of the formula. A SAT solver cannot handle this directly, because the formula is not a true boolean formula: it makes use of additional syntax for embedding arbitrary Smten expressions.

The syntax for the augmented form of boolean formula is shown in Figure 4-6.

$$
\begin{array}{rrcl}
 & v & \in & \textit{Boolean Variable} \\
\text{Formula} & \phi_s & ::= & \texttt{true} \mid \texttt{false} \mid v \mid \neg\phi_s \mid \phi_{s1} \wedge \phi_{s2} \mid \phi_{s1} \vee \phi_{s2} \\
 & & \mid & \texttt{ite}\ \phi_{s1}\ \phi_{s2}\ \phi_{s3} \mid \rho(s)
\end{array}
$$

Figure 4-6: Syntax of boolean formulas augmented with Smten thunks.

The problem of handling nontermination in construction of Smten set expressions reduces to the problem of determining the satisfiability of a ϕ_s formula.

4.3.2 Approaches to Satisfiability of Incomplete Formulas

Before presenting our abstraction-refinement approach to solving satisfiability for ϕ_s formulas, we will review possible alternative approaches.

Evaluate All Thunks

The most basic approach to satisfiability of ϕ_s is to completely evaluate all of the thunks. Without thunks, the syntax of ϕ_s is the same as the syntax of ϕ, and a SAT solver can be used.

This approach is simple, but it works only if all of the thunks can be completely evaluated. This approach would not work on our example.

It may seem silly to suggest this approach as a viable solution to the problem, but it is the approach taken by most of the current attempts at making SAT and SMT solvers easier to use with functional languages. Domain specific embedded languages for SMT fully evaluate all paths in order to construct a SAT formula before sending the formula to the solver. If any path fails to terminate, the program will never finish constructing the formula. Rosette [56] also takes this approach, requiring all programs to be *self-finitizing*, which requires the user to manually bound all recursions, involving pervasive changes to the search-space descriptions.

Use SAT Solver to Determine Unreachable Thunks

We already alluded to this approach. A SAT solver can be used to determine which parts of the ϕ_s formula are unreachable and prune those away. The remaining thunks

are completely evaluated.

Recall the ϕ_s from our previous example:

$$\texttt{ite}\ (x \wedge \neg x)\ \rho(\texttt{fix}\ (\lambda y.y))\ \texttt{true}$$

If the formula $(x \wedge \neg x)$ is satisfiable, it means the `true` branch of the `ite` is reachable, so it should be evaluated, otherwise it can be pruned. If the formula $\neg(x \wedge \neg x)$ is satisfiable, it means the `false` branch of the `ite` is reachable, so it should be evaluated, otherwise it can be pruned.

Using a SAT solver for these queries, we will find that $(x \wedge \neg x)$ is unsatisfiable, and the `true` branch can be pruned away, resulting in the SAT formula `true`.

For a more complex ϕ_s, the entire formula would have to be traversed, with two SAT queries issued for each `ite` condition. This would be prohibitively expensive.

Note that this approach will fail to terminate if there are reachable nonterminating sub-terms of the formula, because they will never be pruned.

Basic Abstraction Refinement

The abstraction-refinement technique approximates the ϕ_s formula using a SAT formula. The results of the SAT query will either provide a result for the ϕ_s formula, or indicate that the approximation needs to be refined.

For example, the formula $\texttt{ite}\ (x \wedge \neg x)\ \rho(\texttt{fix}\ (\lambda y.y))\ \texttt{true}$ can be abstracted with the boolean formula:

$$\texttt{ite}\ (x \wedge \neg x)\ w\ \texttt{true}$$

The fresh boolean variable w is used in place of $\rho(\texttt{fix}\ (\lambda y.y))$. This approximation of the original formula can be solved with a SAT solver, because it does not contain any thunks.

If the SAT solver says the approximation of the formula is satisfiable, the satisfying assignment can be checked against the original formula to see if it is also a satisfying assignment there. If so, the assignment can be returned, otherwise, the approximation must be refined. If the SAT solver says the approximation of the for-

mula is unsatisfiable, then the original formula could be considered unsatisfiable as well, because no possible value for the thunk would lead to a satisfying assignment.

In our example, because the approximated part of the formula is unreachable, any answer returned by the SAT solver for the approximation will hold for the original formula, which means the approximation is good.

An important benefit of the abstraction-refinement approach is it supports arbitrary, reachable, infinite recursion in search-space descriptions. If there is a satisfying assignment that does not depend on the nonterminating part of the search-space description, and the SAT solver returns that assignment, the search will terminate with a useful result. It may be possible the SAT solver returns an assignment that corresponds to a nonterminating computation in the original formula. This case needs to be treated carefully to ensure search still terminates. If the SAT solver says the approximation is unsatisfiable, search may terminate when semantically it should fail to terminate. We generally feel this is acceptable behavior, even if not semantically justified.

Improved Abstraction Refinement

One problem with the basic abstraction refinement approach is that there is nothing to stop the SAT solver from repeatedly returning assignments that require the approximation to be refined. One insight from Leon [54] is that the SAT queries can include additional information that directs the SAT solver to find a satisfying assignment requiring no refinement of the approximation.

For our example, in addition to using the approximation `ite` $(x \wedge \neg x)$ w `true`, we can generate another term that indicates when the approximation is valid. In this case, the approximation is valid if the `true` branch is not taken, so the additional term would be $\neg(x \wedge \neg x)$. We will call the approximation of the formula ϕ_a and the formula indicating the approximation's validity ϕ_v:

$$a = \texttt{ite}\ (x \wedge \neg x)\ w\ \texttt{true}$$
$$v = \neg(x \wedge \neg x)$$

If the SAT formula $\phi_a \wedge \phi_v$ is satisfiable, the satisfying assignment must be a satisfying assignment to the original formula, because the approximation is satisfied and is a valid approximation. If the SAT formula ϕ_a is unsatisfiable, the original formula can be considered unsatisfiable (with relaxation to the semantics discussed above). Otherwise the approximation can be refined and the procedure repeated.

Smten Abstraction Refinement

The Smten abstraction refinement approach is a variation on the improved abstraction refinement approach that avoids introducing free variables for approximated terms. This is valuable both because adding free variables increases the time to solve the queries, and because our approach can determine if a query is unsatisfiable without having to relax the semantics.

Instead of introducing free variables for approximated terms, we can use arbitrary values, because we will only use the results from the approximation when we know the approximation is valid. For example, we can arbitrarily approximate the thunk as `true`:

$$\phi_a = \texttt{ite}\ (x \wedge \neg x)\ \texttt{true}\ \texttt{true}$$
$$\phi_v = \neg(x \wedge \neg x)$$

If the SAT formula $\phi_a \wedge \phi_v$ is satisfiable, the satisfying assignment must be a satisfying assignment to the original formula, because the approximation is satisfied and is a valid approximation.

Otherwise, we check if $\neg\phi_v$ is satisfiable. If not, then it is not possible for the

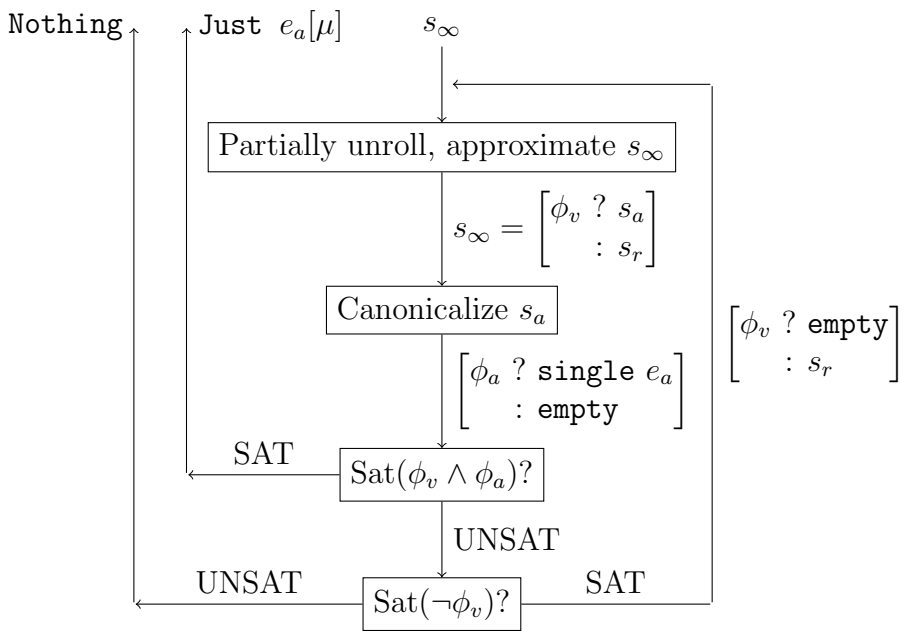

Figure 4-7: The Smten abstraction refinement procedure.

approximation to be invalid, so there must be no satisfying assignment to the original formula.

Otherwise we refine the approximation, taking into account that we have learned there are no satisfying assignments to the original formula when ϕ_v is true, so ϕ_v must be false.

Notice that the approximation formula ϕ_a we use can be simplified, because both branches of the `ite` are true:

$$\phi_a = \texttt{true}$$

$$\phi_v = \neg(x \wedge \neg x)$$

Because we can choose arbitrary values for the approximated terms, the approximation ϕ_a can often be simplified greatly, as in this case, which further reduces the time to construct and solve the SAT query.

Figure 4-7 represents the Smten abstraction refinement procedure graphically in terms of search spaces instead of boolean formulas. A search space s_∞, which may

contain nonterminating subterms, is partially unrolled and partitioned into those parts of the search space that have been fully unrolled, represented with s_a, and those parts of the search space that could be unrolled more, represented with s_r. As in our previous description, the formula ϕ_v represents the conditions under which the search space s_∞ is the same as s_a.

The finite search space s_a can be reduced to a set expression of the form:

$$\begin{bmatrix} \phi_a \ ? \ \texttt{single}\ e_a \\ \quad : \ \texttt{empty} \end{bmatrix}$$

If the formula $\phi_v \wedge \phi_a$ is satisfiable with assignment μ, then $e_a[\mu]$ is an element of the original search space s_∞. Otherwise, if the approximation s_a is equivalent to s_∞, then $\neg\phi_v$ will be unsatisfiable, and there are no elements in s_∞. If that is not the case, then the procedure repeats using the refined search space:

$$\begin{bmatrix} \phi_v \ ? \ \texttt{empty} \\ \quad : \ s_r \end{bmatrix}$$

This takes advantage of the fact that we have already searched all of s_a, so it must be empty.

4.4 Optimizing the Smten Approach

Many of the optimizations required for an application developed with Smten to work well in practice are specific to the application and can be expressed in the user's code. There are a handful of important optimizations, however, that are built into Smten. Whereas user-level optimizations lead to changes in the high-level structure of the generated queries, the optimizations built into Smten focus on reducing the cost of generating those queries. This is achieved primarily by exploiting sharing and pruning parts of the query that have no effect.

Characterizing the impact of these optimizations is difficult. Combinatorial search

problems, by their nature, are sensitive to scaling: small changes to the implementation can affect performance by orders of magnitude. In our experience, this is the difference between a tool that works well in practice and one that fails to work entirely. Nevertheless, we attempt to describe the broad impacts of our optimizations in this section and present some empirical results in Section 6.4.

4.4.1 Normal Forms for Partially Concrete Expressions

The rules *st-beta-phi*, *st-fst-phi*, *st-snd-phi*, and *st-case-phi* for pure evaluation of partially concrete expressions push their corresponding primitive operations inside the branches of a ϕ-conditional expression. This duplicates work in query generation and leads to redundancy in the generated query. We can eliminate this duplication by normalizing partially concrete expressions to share structure in subexpressions so that the primitive operations are evaluated only once. The consequence of normalization is that none of the rules that duplicate primitive operations will ever be applicable.

Products The normal form for a partially concrete product is a concrete product with partially concrete components:

$$\begin{bmatrix} \phi \ ? \ (e_{11}, e_{12}) \\ \quad : \ (e_{21}, e_{22}) \end{bmatrix} \rightarrow (\ \begin{bmatrix} \phi \ ? \ e_{11} \\ \quad : \ e_{21} \end{bmatrix} \ , \ \begin{bmatrix} \phi \ ? \ e_{12} \\ \quad : \ e_{22} \end{bmatrix} \ )$$

This eliminates the need for *st-fst-phi* and *st-snd-phi*.

Booleans The normal form for a boolean expression is a ϕ-conditional expression whose left branch is `True` and right branch is `False`: ϕ `?` `True` `:` `False`. As a shorthand, we use ϕ to represent the normal form of the boolean expression, and apply the following rewrite:

$$\phi \ ? \ \phi_1 \ : \ \phi_2 \rightarrow \texttt{ite} \ \phi \ \phi_1 \ \phi_2$$

In effect, Smten boolean expressions are translated to boolean formulas and handled

directly by the SAT solver. An analogous approach can be used for types with direct support in the SMT backend, as discussed in Section 4.4.3.

Sums Sum types can be viewed as a generalization of the boolean case. They are reduced to the canonical form $\phi \ ?\ \texttt{inl}\ e_l\ :\ \texttt{inr}\ e_r$ using:

$$\begin{bmatrix} \phi \ ?\ (\phi_1 \ ?\ \texttt{inl}\ e_{l1}\ :\ \texttt{inr}\ e_{r1}) \\ \quad :\ (\phi_2 \ ?\ \texttt{inl}\ e_{l2}\ :\ \texttt{inr}\ e_{r2}) \end{bmatrix} \rightarrow \begin{bmatrix} \texttt{ite}\ \phi\ \phi_1\ \phi_2 \ ?\ (\texttt{inl}\ \phi \ ?\ e_{l1}\ :\ e_{l2}) \\ \qquad\qquad\qquad\qquad :\ (\texttt{inr}\ \phi \ ?\ e_{r1}\ :\ e_{r2}) \end{bmatrix}$$

This allows us to replace the *st-case-phi* rule with one that does not duplicate the work of the case expression:

$$\texttt{case}\ \begin{bmatrix} \phi \ ?\ \texttt{inl}\ e_1 \\ \quad :\ \texttt{inr}\ e_2 \end{bmatrix}\ f_1\ f_2 \rightarrow \begin{bmatrix} \phi \ ?\ (f_1\ e_1) \\ \quad :\ (f_2\ e_2) \end{bmatrix}$$

Functions The normal form for functions are functions:

$$\begin{bmatrix} \phi \ ?\ f_1 \\ \quad :\ f_2 \end{bmatrix} \rightarrow \lambda x\ .\ \begin{bmatrix} \phi \ ?\ (f_1\ x) \\ \quad :\ (f_2\ x) \end{bmatrix}$$

This eliminates the need for *st-beta-phi* and creates additional opportunities for sharing when the same function is applied multiple times.

4.4.2 Leverage Unreachable Expressions

There are many places in the implementation where unreachable expressions are created. For example, the rule *sx-empty* reduces `empty` to $\{\bot \mid \texttt{false}\}$, where $\bot$ could be any value because it is unreachable. By explicitly tracking which expressions are unreachable, we can simplify our queries before passing them to the solver. For instance, $\phi \ ?\ e\ :\ e_x$ can be simplified to e if e_x is known to be unreachable, because if e_x is unreachable, ϕ must be `true` for every assignment.

Recall that our scheme to avoid spurious nontermination (Section 4.3) introduces arbitrary values for approximated subterms. Using explicitly unreachable expressions

instead of an arbitrary value has the effect of selecting the value that simplifies the approximation the most.

4.4.3 Exploit Theories of SMT Solvers

Smten can naturally leverage SMT solvers by expanding the syntax of formulas for types directly handled by the solver and using a formula as the canonical representation for these types as we do for booleans. For example, consider the following boolean expression involving integers:

$$\begin{bmatrix} \phi_1 \ ?\ 1 \\ \quad :\ 2 \end{bmatrix} < \begin{bmatrix} \phi_2 \ ?\ 2 \\ \quad :\ 3 \end{bmatrix}$$

In order to reduce this expression to the canonical form for expressions of type `Bool`, the primitive `<` operator must be pushed into the branches of the ϕ-conditional expressions:

$$\begin{bmatrix} \phi_1 \ ?\ (\phi_2 \ ?\ (1 < 2)\ :\ (1 < 3)) \\ \quad :\ (\phi_2 \ ?\ (2 < 2)\ :\ (2 < 3)) \end{bmatrix}$$

This exponential duplication of work can be avoided, assuming we have an SMT solver that supports a theory of integers, by constructing a formula using the SMT solver's `lt` operator for the comparison. The result is an SMT formula:

$$\texttt{lt}\ (\texttt{ite}\ \phi_1\ 1\ 2)\ (\texttt{ite}\ \phi_2\ 2\ 3)$$

To fully exploit an underlying theory it is important to have direct access to unconstrained variables of the corresponding types. For booleans, direct access to unconstrained variables can be achieved using existing search-space primitives:

```
free_Bool :: Space Bool
free_Bool = union (single True) (single False)
```

There is no direct way to introduce a free variable of a different type, however.

For this reason, we add new search primitives that create free variables directly, if supported by the SMT solver. For example, for theories of integers and bit vectors, we introduce the primitives:

```
free_Integer :: Space Integer
free_Bit     :: Space (Bit n)
```

Consider the query generated for the following search space:

```
do x ← free_Bit2
   guard (x > 1)
```

The `free_Bit2` function could be implemented in terms of existing `Space` primitives:

```
free_Bit2 :: Space (Bit 2)
free_Bit2 = do
  x0 ← union (single 0) (single 1)
  x1 ← union (single 0) (single 1)
  single (bv_concat x0 x1)
```

This would lead to a generated SMT query such as:

$$(\texttt{bv\_concat}\ (\texttt{ite}\ x_0\ 0\ 1)\ (\texttt{ite}\ x_1\ 0\ 1)) > 1$$

If `free_Bit` is provided as a primitive, however, then `free_Bit2` can use it directly as the implementation. This leads to a simpler generated SMT query: $x > 1$, where x is a free bit-vector variable in the SMT query.

Providing `free_Bit` and `free_Integer` requires modifying the implementation of Smten. This is required only for primitive Smten types. Theories that can be expressed in terms of currently supported SMT theories in Smten can be expressed directly in the Smten language by composition of the Smten primitive theories.

We will note the same method for approximation used for boolean SAT formulas discussed in Section 4.3.2 can be extended to SMT formulas. Now, instead of approximating each formula using boolean formulas ϕ_v and ϕ_a, a formula of a specific

type, such as integer or bit-vector, is approximated using a boolean formula ϕ_v, and formulas of the same integer or bit-vector type for ϕ_a.

Note that is it is possible for the primitive `free_Integer` to violate the strictness of search expected for free integers. This is because the SMT solver can determine a query involving free integers is unsatisfiable when the Smten semantics would require infinite computation to say the same thing. We believe this is an acceptable relaxation of the semantics, because it can give only more information than a semantically correct implementation.

4.4.4 Formula Peep-Hole Optimizations

We perform constructor oriented optimizations to simplify formulas as they are constructed. In each of the following examples, the evaluation of lazily applied functions for constructing the formula ϕ_x can be entirely avoided:

$$\phi_x \wedge \texttt{false} \rightarrow \texttt{false}$$

$$\texttt{ite false } \phi_x\ \phi_y \rightarrow \phi_y$$

$$\texttt{ite } \phi_x\ \phi_y\ \phi_y \rightarrow \phi_y$$

4.4.5 Sharing in Formula Generation

Consider the user-level Smten code:

```
let  y = x + x + x
     z = y + y + y
in z < 0
```

It is vital we preserve the user-level sharing in the generated query, rather than generating the query:

$$((x + x + x) + (x + x + x) + (x + x + x)) < 0$$

To prevent this exponential growth, we are careful to maintain all of the user-level sharing in the query when passed to the solver, including dynamic sharing that occurs due to user-level dynamic programming, memoization, and other standard programming techniques.

Chapter 5

Smten Language Compiler

To evaluate the Smten approach to search, we have developed a fully featured compiler for the Smten language, as well as a language runtime and standard libraries. Combined, the compiler, runtime, and standard libraries we developed serve as a practical implementation of the Smten search approach.

Figure 4-1 from Section 4.2 shows the augmented syntax of KerSmten terms used in the syntax-directed approach to Smten search. The syntax includes terms from a simply typed lambda calculus, the search-space primitives, and the ϕ-conditional and set expressions. To implement the full Smten language, we use Haskell in place of the simply typed lambda calculus. The search-space primitives and set expressions can be supplied in the form of Haskell libraries. The ϕ-conditional expression, however, cannot be supplied as a library, because it affects every type of expression in the Haskell language, including primitive data types. Instead, we must replace every Haskell type with a new canonical form that supports embedded SMT formulas, the ϕ-conditional operation, and all the original primitive operations for that type.

This chapter gives a brief overview of the Smten compiler. Section 5.1 describes why we provide a compiler for Smten rather than just a set of Haskell libraries. Section 5.2 describes how we reuse the Glasgow Haskell Compiler to implement the Smten compiler. In Section 5.3 we review Smten's canonical forms for the Haskell datatypes. Section 5.4 discusses how we implement partial unrolling of expressions for the Smten abstraction-refinement procedure. Section 5.5 discusses how we treat

the solver as an oracle for search to compute the concrete element returned by search, and Section 5.6 discusses how sharing in formula generation is implemented.

5.1 Why Smten Needs a Compiler

In principle, we could supply a set of libraries for Haskell developers to use to implement Smten-based combinatorial search applications. The libraries would include the `Space` type, search primitives, set expression, and alternate definitions for every primitive Haskell type that support the ϕ-conditional expression. It is better, however, to provide a compiler for a Smten language than reuse Haskell as is.

Without a compiler for the Smten language, users of Smten would have to have detailed knowledge of the Smten approach to define new user data types that efficiently support the ϕ-conditional expression. They would also have to know how Haskell's primitive data types have been rewritten. With a compiler for the Smten language, these details can be codified in the implementation of the compiler. A user would not have to understand the underlying Smten approach, and they could benefit from future improvements in the Smten approach simply by switching to the latest compiler, rather than having to reimplement their application.

The Haskell language does not have sufficient support for reflection to implement the Smten approach as a user library. There is no way to access the structure of arbitrary user-defined functions, and though it provides some mechanisms for metaprogramming with Template Haskell [47], in practice working with Template Haskell has significant limitations.

A domain specific embedded language could be developed for the Smten approach in Haskell, but using a domain specific embedded language for Smten would suffer similar problems as using domain specific embedded languages for SAT and SMT: the user would have to transform their data-types and constraints into Smten data-types and constraints manually.

For these reasons, we developed a Smten compiler that automatically compiles Haskell programs with search primitives to use the Smten search approach.

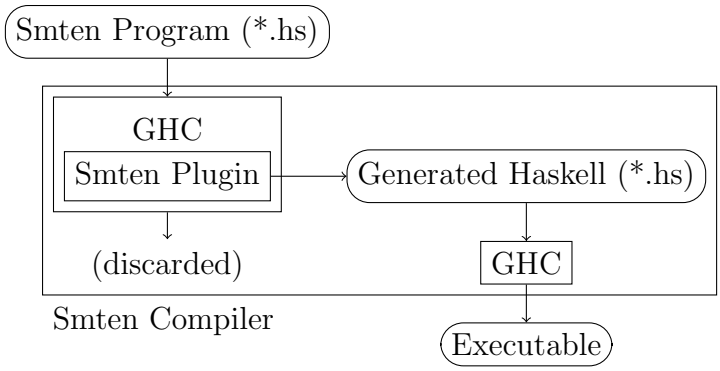

Figure 5-1: Organization of the Smten compiler.

5.2 General Compilation Approach

The task of the Smten compiler is to rewrite users' Haskell functions to use alternate implementations of the Haskell data types that support the ϕ-conditional expression. User-defined data types are also rewritten to support the ϕ-conditional expression.

Rather than implementing a compiler for Smten from scratch, we can reuse the Glasgow Haskell Compiler (GHC), both for front-end and back-end compilation. The front-end of the Smten compiler reuses GHC's package management, parser, type inference, type checking, and module support. The back-end of the Smten compiler reuses GHC's code generation and runtime.

The structure of the Smten compiler is shown in Figure 5-1. We have implemented a GHC plugin to modify the behavior of GHC. Smten compilation consists of two passes. The first pass runs GHC with the Smten plugin, intercepting the core intermediate representation from the GHC compiler and generating a new Haskell program that implements the Smten search approach for the original program. The normal output from GHC in this phase is discarded. The second pass runs GHC normally on the generated Haskell program to produce an executable.

Figure 5-2 shows a breakdown of the lines of code used to implement the Smten compiler. The Smten plugin is responsible for generating new data type definitions and function implementations for each user-defined data type and function. The runtime includes the definitions of the `Space` type, search-space primitives, set expressions, and versions of the primitive Haskell data types that support the ϕ-conditional

Component	Lines of Haskell
Plugin	1.6K
Runtime	3.0K
Solver Interfaces	2.0K
Standard Libraries	4.0K
Test Cases	1.7K
Total	12.3K

Figure 5-2: Breakdown of lines of code for Smten compiler implementation.

expression. The Smten runtime currently supports five different solvers for use in search: Yices1, Yices2, STP, Z3, and MiniSat. The Smten compiler is open source and available at `http://github.com/ruhler/smten`.

5.3 Data Type Representations

In this section we review the canonical representation for the Smten primitive types, which support the ϕ-conditional expression.

5.3.1 Booleans

The boolean data type is treated as a primitive type in Smten, using an SMT formula for the underlying representation. The syntax for an SMT formula used by Smten consists of standard boolean operations and the SMT background theories of unbounded integers with linear arithmetic and fixed-width bit vectors. The ϕ-conditional expression constructs a formula with `ite`.

5.3.2 Integer

The Integer type is represented using an SMT formula. The ϕ-conditional expression constructs an integer formula with `ite`.

5.3.3 Bit

Smten includes a primitive Bit type for bit-vectors, which is not present in Haskell. The primitive bit-vector type is represented using an SMT formula, and the ϕ-conditional expression constructs a bit-vector formula with `ite`.

5.3.4 Int

The primitive Int type is represented as a map from Haskell Int to boolean formula. For example, the Smten expression $(\phi_1\ ?\ (\phi_2\ ?\ 1\ :\ 2)\ :\ 3)$ is represented using the map:

$$\begin{aligned} 1 &\mapsto \phi_1 \wedge \phi_2 \\ 2 &\mapsto \phi_1 \wedge \neg\phi_2 \\ 3 &\mapsto \neg\phi_1 \end{aligned}$$

The ϕ-conditional expression joins the boolean formulas for each field using `ite`.

5.3.5 Char

The primitive Char type is represented using the same implementation as the Int data type.

5.3.6 Function

Smten functions are represented as Haskell functions. The ϕ-conditional expression is implemented as:

$$\phi\ ?\ f_1\ :\ f_2 \quad = \quad \lambda x\ .\ \phi\ ?\ (f_1\ x)\ :\ (f_2\ x)$$

5.3.7 IO

The IO type is represented as the Haskell IO type. IO operations cannot affect the formula constructed in a search space. Because of the way we compute concrete

elements of a search space (discussed in Section 5.5), we do not need to provide support for ϕ-conditional expressions of type IO.

5.3.8 Algebraic Data Types

Algebraic data types, including user-defined algebraic data types, are represented as a mapping from constructor to a boolean formula and the constructor's arguments. This is similar to the way the Int type is represented, except that every constructor is present in the map, and the arguments to the constructor are included in the map.

For example, the Smten expression $(\phi_1 \ ? \ (\phi_2 \ ? \ \texttt{Just}\ a \ : \ \texttt{Nothing}) \ : \ \texttt{Just}\ b)$ is represented using the map:

$$\begin{aligned} \texttt{Just} &\mapsto (\phi_1 \wedge \phi_2) \vee \neg\phi_1, \quad \phi_1 \wedge \phi_2 \ ? \ a \ : \ b \\ \texttt{Nothing} &\mapsto \phi_1 \wedge \neg\phi_2 \end{aligned}$$

Haskell case expressions are rewritten to ϕ-conditional expressions, using the map to quickly access the formula and arguments for each constructor.

5.4 Partial Unrolling for Abstraction Refinement

The abstraction refinement procedure presented in Section 4.3.2 uses heuristics to decide how much a search-space description is unrolled. The heuristics in the actual Smten runtime unroll a computation completely or until it raises an exception.

To implement this heuristic, we use GHC's exception handling mechanisms. The heuristic quickly detects explicit user errors, such as uses of the `error` primitive, floating point exceptions, and pattern match failures. The heuristic does not otherwise consider infinite computations as long running. As a consequence, infinite computations in search-space descriptions lead to nontermination in the implementation of search.

One reason this behavior is valuable is because infinite computations in search are often opportunities to improve the search description. Rather than relying on heuristics to detect when something is unreachable, it can be expressed directly,

leading to much more efficient code.

We expect that better heuristics can be developed in the future to help developers less experienced in finding and fixing these infinite computations build working applications. For the advanced developer, it is probably desirable to have the option to report infinite computations as potential performance opportunities to be investigated.

5.5 SAT as an Oracle

The description of the syntax-directed approach to Smten search given in Section 4.2 traverses the body of the set expression to produce a concrete search result based on the satisfying assignment from the SAT or SMT formula.

In practice we use a different approach that treats the assignment from the solver as an oracle. Using this approach, the search space is evaluated twice. The first time the search space is evaluated, the formula ϕ is produced, but the body of the set expression is discarded. If the formula ϕ is satisfied, then the search space is re-evaluated directly, using the assignment from the SAT solver as an oracle.

The main benefit of this approach is that we do not need a special way to support ϕ-conditional expressions for data types that cannot affect the formula ϕ, such as the IO type.

5.6 Implementing Preservation of Sharing

The Smten implementation constructs an abstract syntax tree for an SMT formula, then converts the abstract syntax tree into back-end solver interface calls to transmit the formula to the solver. In reality, the abstract syntax tree is a directed acyclic graph (DAG), because nodes of the formula can be reused in different parts of the formula. Rather than duplicating the node and all its children, we share the node.

An important optimization in Smten is to preserve this sharing when transmitting the formula to the solver. Each shared node should be transmitted to the solver only

a single time, not once for each use. Otherwise the same node could be re-transmitted exponentially many times.

Preservation of sharing is more complicated to implement in a pure functional language like Haskell than an imperative language like C, because object pointers are not immediately accessible. The most efficient implementation we have found for preservation of sharing is to allocate an `IORef` for each node in a formula that caches the value of the node after transmission to the back-end solver. This approach requires allocating an `IORef` for each node and checking it to see if the node has already been transmitted to the back-end solver or not when traversing the abstract syntax tree. This works much better than using stable names [45] and a hash table to keep track of visited nodes, because the hash table grows very large and contains mostly unshared expressions.

Chapter 6

Evaluation of Smten Search

To evaluate Smten, we used it to develop three complex satisfiability-based search applications: a reimplementation of the HAMPI string constraint solver [31], a model checker based on k-induction, and a reimplementation of the sketch tool for program synthesis [50].

In this chapter we compare our implementations against state-of-the-art SAT and SMT-based tools. We show that our Smten-based implementations were relatively easy to develop, use many fewer lines of source code than the original implementations, and perform as well as the original implementations. After presenting our string constraint solver (Section 6.1), model checker (Section 6.2), and program synthesis tool (Section 6.3), we present some results demonstrating the importance of optimizations to the Smten search approach (Section 6.4) and discuss our experience using a functional language for Smten (Section 6.5).

The source code for our implementation of Smten and each of the applications can be found on github at `http://github.com/ruhler/` in the `smten` and `smten-apps` repositories respectively.

6.1 Hampi String Constraint Solving

SHAMPI is a Smten-based reimplementation of HAMPI, a string constraint solver whose constraints express membership of strings in both regular languages and context-

Module	Lines	Description
CFG, SCFG, RegEx, Hampi	356	Data type definitions
Fix	103	Fix-sizing
Match	50	Core match algorithm (memoized)
Lexer, Grammar	303	Parser for HAMPI constraints
SChar	90	Char representations: Integer, Bit, Int, Char
Query	90	Query orchestration
Main	83	Command line parsing
Total	1059	Lines of Smten

Figure 6-1: Breakdown of lines of code in SHAMPI.

free languages.

A HAMPI input consists of the definitions for regular expressions and context free grammars (CFGs), bounded-size string variables, and constraints on strings described in terms of the regular expressions, grammars, and string variables. The output from HAMPI is a string that satisfies the constraints or a report that the constraints are unsatisfiable.

The HAMPI tool has been applied successfully to testing and analysis of real programs, most notably in static and dynamic analyses for SQL injections in web applications and automated bug finding in C programs using systematic testing. HAMPI's success is due in large part to preprocessing and optimizations that recognize early when a CFG cannot match a string, regardless of the undetermined characters the string contains.

The original version of HAMPI is implemented in about 20K lines of Java and uses the STP [22] SMT solver. In contrast, our implementation of SHAMPI is around 1K lines of Smten code. This is an order of magnitude reduction in code size.

Figure 6-1 shows a breakdown of the lines of code in our implementation of SHAMPI. The core match algorithm, optimized with user-level memoization, requires only 50 lines of code and query orchestration only 90 lines of code. In contrast, the parser for the string constraint specifications is 300 lines of code, which reflects that when using Smten, most of the development effort is spent implementing generic procedures specific to the application, and not optimizing query construction.

Our initial development of SHAMPI required approximately three weeks of effort,

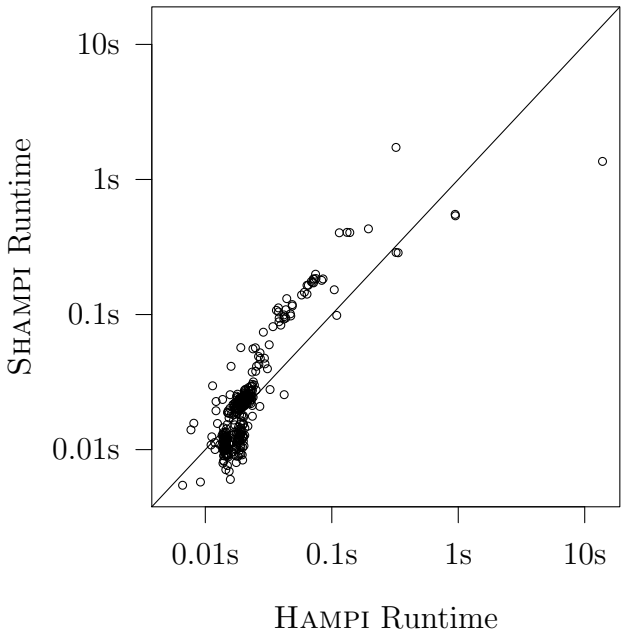

Figure 6-2: Performance of HAMPI and SHAMPI with STP.

including efforts required to understand the tool and important optimizations to implement. We revisited the implementation a year after the initial development and were happy to find we could still understand the code and were able to identify and implement some additional optimizations easily.

To evaluate the performance of SHAMPI, we ran both HAMPI and SHAMPI on all benchmarks presented in the original HAMPI paper. We experimented with two different backend solvers for SHAMPI: STP, which is the same backend solver used by HAMPI, and the Yices2 [64] SMT solver, which performs notably better. Figure 6-2 and Figure 6-3 compare the performance of HAMPI and SHAMPI for each of the benchmarks. Those points below the 45 degree line are benchmarks on which SHAMPI out-performs the original HAMPI implementation. For both SHAMPI and HAMPI, we took the best of 8 runs. SHAMPI was compiled with GHC-7.6.3 [23]. We ran revision 46 of a single HAMPI server instance for all runs of all tests on HAMPI to amortize away the JVM startup cost; this allows just-in-time compilation optimization, which biases the results towards HAMPI slightly.

Assuming our implementation includes the same algorithms and optimizations

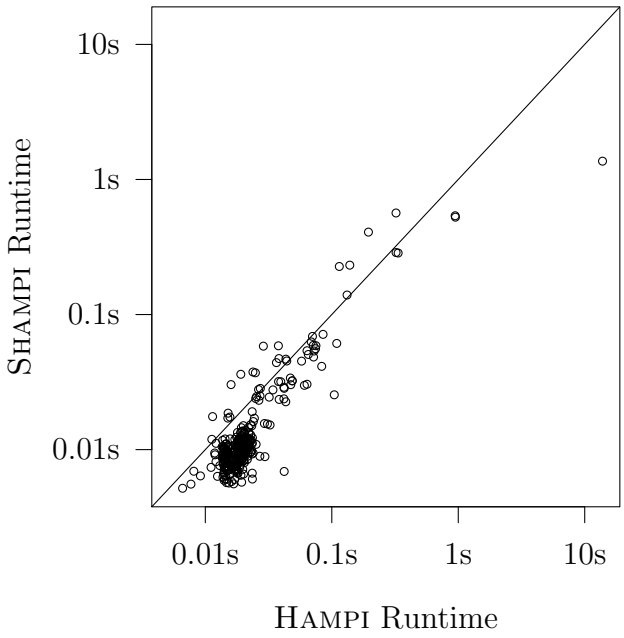

Figure 6-3: Performance of HAMPI and SHAMPI with Yices2.

for generating the queries, as we believe to be the case, Figure 6-2 represents the overheads associated with using the more general Smten framework for solving the string constraint problem. Figure 6-3 represents the benefits the framework provides in requiring a trivial amount of effort to experiment with different backend solvers.

6.2 Hardware Model Checking

Hardware model checking is used to verify properties of hardware systems. The hardware is modeled using a finite state transition diagram. We have implemented Saiger, a model checker that supports the input format and property checking required by the 2010 hardware model checking competition [28]. Saiger tests whether any bad states are reachable from an initial state.

Saiger was implemented using a direct transliteration of the k-induction approach to model checking described in [48]. The core k-induction algorithm is described in less than 100 lines of Smten, with an additional 160 or so lines for parsing and evaluation of hardware described in the Aiger format, around 50 lines to adapt the

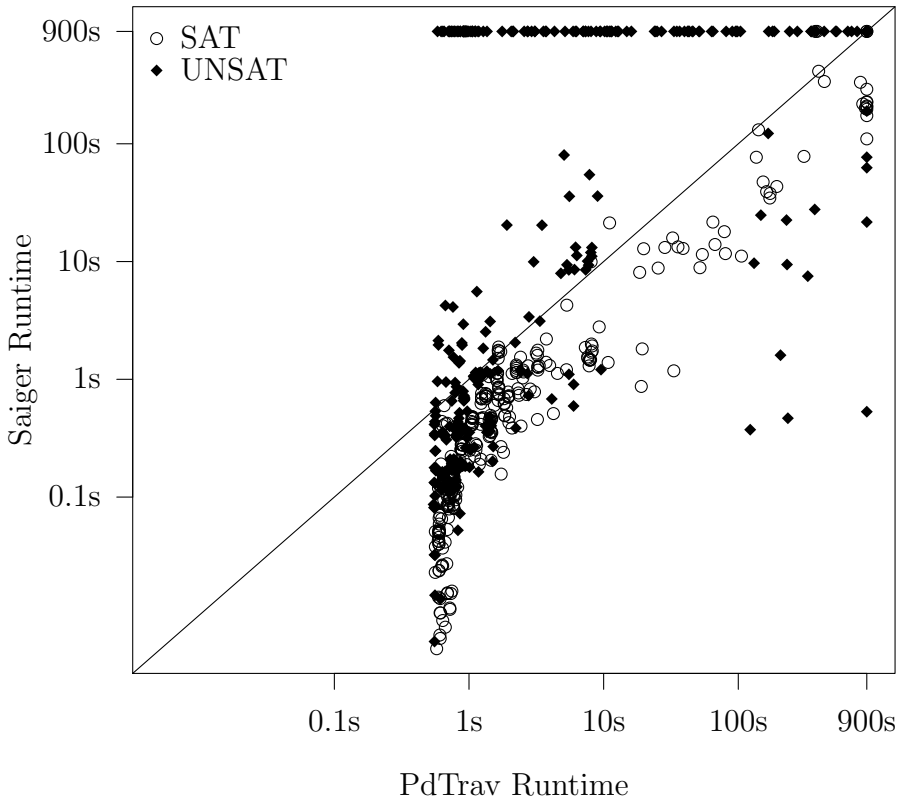

Figure 6-4: Performance of PdTrav and Saiger.

Aiger format to the core algorithm, and approximately 70 lines for parsing options and calling the core algorithm. In total, implementing Saiger takes less than 400 lines of Smten code, and took on the order of one week of effort to implement. The majority of that effort was devoted to understanding and implementing the Aiger format for hardware models; little effort was spent implementing or optimizing the k-induction algorithm.

Figure 6-4 shows the performance of Saiger on the benchmarks from the 2010 hardware model checking competition compared to PdTrav [7]. PdTrav is the best model checker in the 2010 competition that we could run ourselves able to identify both models with counter-examples exhibiting bugs, called SAT, and models without counter-examples exhibiting bugs, called UNSAT. Those points above the 45

degree line are cases on which version 3.2.0 of PdTrav out-performs Saiger. Those points below the 45 degree line are cases on which Saiger out-performs PdTrav. For a large majority of the benchmarks, Saiger out-performs PdTrav, especially on those benchmarks for which counter-examples exhibiting bugs were found, labeled as SAT benchmarks. There are a number of UNSAT benchmarks on which our model checker exceeds the 900 second timeout limit that PdTrav is able to solve. This is almost certainly due to PdTrav using an interpolant-based approach to model checking, which can result in significantly smaller search spaces in proving unsatisfiability over the straightforward k-induction algorithm we implemented. While we have not investigated it thoroughly, we are confident that implementing such an algorithm in Smten would yield similar results.

Saiger demonstrates the ease with which complex applications can be described in Smten and perform well. The code for the core algorithm matches the descriptions from the paper presenting the algorithm; it is easy to understand and modify. Additionally, the core model checking algorithm is completely independent from the form used to represent the hardware model. The core algorithm could be reused unmodified for many different models, and perhaps even for software model checking.

6.3 Sketch Program Synthesis

To showcase Smten's ability to express complex multi-query computations we implemented the core subset of the Sketch [50] program synthesis tool. Sketch is a state-of-the-art language and SAT-based tool that takes as input a sketch of a program where some expressions are replaced with holes, and a specification of the program's desired behavior. The Sketch tool outputs a complete program filling the provided holes such that it meets the specification given.

For example, Figure 6-5 shows a sketch of an optimized function for isolating the rightmost unset bit of a word, along with an unoptimized specification. Each token `??` represents a hole for the synthesizer to fill in. The output of running the sketch tool on this example is a synthesized optimized function:

```
bit[W] isolate0 (bit[W] x) {
  bit[W] ret=0;
  for (int i = 0; i < W; i++) {
     if (!x[i]) {
        ret[i] = 1;
        break;
     }
  }
  return ret;
}

bit[W] i0sketch(bit[W] x) implements isolate0 {
  return ~(x + ??) & (x + ??);
}
```

Figure 6-5: A Sketch program sketch and specification.

```
bit[W] i0sketch (bit[W] x) {
  return ~(x + 0) & (x + 1);
}
```

The core of Sketch is realized by a Counter-Example Guided Inductive Synthesis (CEGIS) procedure [49]. CEGIS decomposes a doubly quantified formula into a sequence of alternating singly quantified queries that can be directly answered by SAT solvers. Sketch has been used in a variety of contexts to predict user preferences [10], optimize database applications [11], and factoring of functionalities [63].

The original Sketch tool was developed over almost a full decade and represents over 100K lines of code (a mix of approximately 86K lines of Java and 20K lines of C++ code). In comparison our implementation, called Ssketch, is only 3K lines of Smten. While our implementation does not support all the features of the Sketch language, *e.g.,* stencils, uninterpreted functions or package management, it does include support for the core language features.

Implementing Ssketch has required greater development effort than both SHAMPI and Saiger, because of the large set of features in the Sketch language. Most of the approximately three to six man-months of effort required for us to implement Ssketch

was devoted to implementing the semantics and features of the Sketch language, not the program synthesis or query generation aspects of Ssketch. The effort is comparable to the effort that would be required to develop a simple, traditional compiler for the Java-like language of Sketch.

To evaluate our implementation of Ssketch against the original implementation of Sketch, we ran both tools on the gallery benchmarks provided in the Sketch distribution. Figure 6-6 shows the results of running our implementation of Ssketch and version 1.6.4 of the original Sketch tool on the gallery benchmarks supported by Ssketch. The number of iterations required for the CEGIS algorithm is sensitive to which counter-examples are returned by the underlying solver. Because of randomness used internally in Sketch, running Sketch repeatedly on the same benchmark can produce very different results. For this reason, we ran the original Sketch 20 times on each benchmark. Ssketch is more repeatable for a given choice of backend solver, but switching to a different solver has the same effect as randomness in the original Sketch implementation. For this reason, we ran our version of Ssketch 8 times on each benchmark with each backend solver supported by Smten. In Figure 6-6, for a given benchmark, the left line and boxes show the runtime for each of the 20 runs of the original Sketch tool on the benchmark. The right line and boxes show the runtime for each of the runs of Ssketch on the benchmark, colored according to the backend solver used. The horizontal bars indicate the median runtime of the tools for given benchmark and solver configurations.

The takeaway from Figure 6-6 is that for these benchmarks, our implementation of Sketch using Smten is comparable in performance to the original version of Sketch, which is remarkable, considering how many fewer lines of code our implementation required, and the years of optimization effort invested in the original implementation of Sketch.

Not all of the gallery benchmarks in the Sketch distribution are supported by our implementation of Ssketch. Three of the benchmarks require uninterpreted functions or stencils, language features not implemented in Ssketch. Nine of the benchmarks use features supported by Ssketch, but fail to terminate. When we first implemented

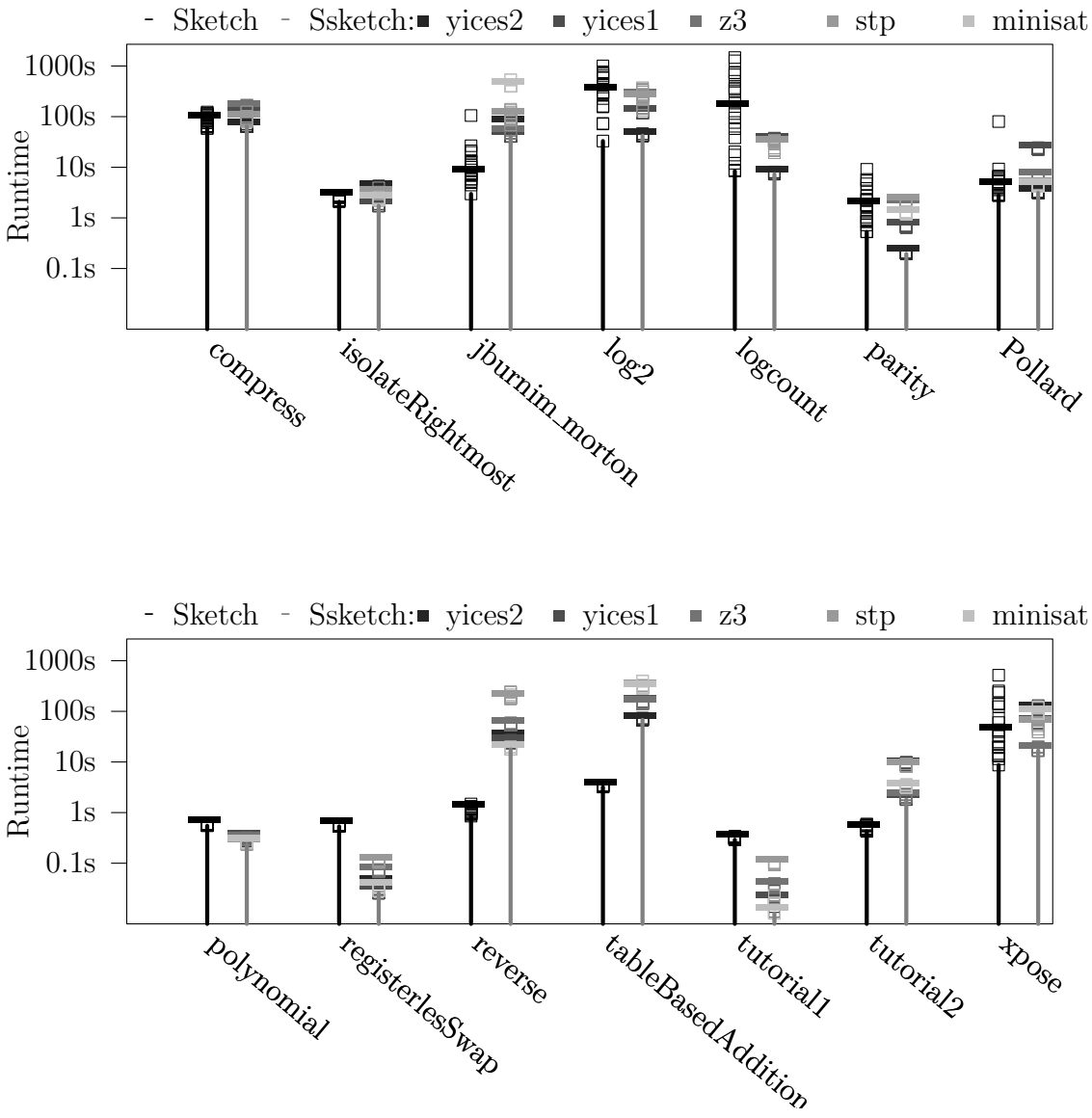

Figure 6-6: Performance of Ssketch and Sketch on gallery benchmarks.

Ssketch, this was the case for all of the benchmarks shown in Figure 6-6, and is typical of development of an SMT-based tool. It is fairly straightforward, though time consuming, to minimize the failing test cases and identify the reason it fails to complete. Once the problem is understood, it is usually simple to modify the Smten code for Ssketch to fix the problem. We expect with further investigation of the failing gallery benchmarks, we could achieve results comparable to Sketch on all the benchmarks.

Examples of performance problems we identified and fixed in our initial version of Ssketch include computing the length of a bulk array access from the original array rather than an array truncated by a partially concrete amount, implementing correct behavior when a Sketch function has no return statement, correctly identifying static bounds on loop unrolling, and using an association list to represent the variable scope rather than a balanced binary tree, whose structure is more sensitive to the order in which variables are updated. All of these problems could be fixed by modifying the implementation of Ssketch; they do not require any changes to the Smten language or implementation.

6.4 Optimizations

It is difficult to quantify the effects of the individual Smten optimizations discussed in Section 4.4. For example, the normal forms for partially concrete expressions are built into our implementation of Smten in a fundamental way. We can, however get a sense of the effects of some of the other optimizations.

Figure 6-7 compares the performance of Ssketch with and without peephole optimizations (which includes pruning based on unreachable expressions), and Figure 6-8 compares the performance of Ssketch with and without sharing optimizations in the Smten implementation. Each point corresponds to the runtime of a different one of the Sketch mini performance tests.

The behavior reflected in these graphs is that a large number of test cases are unaffected by the optimizations, but a fair number of the test cases are *significantly*

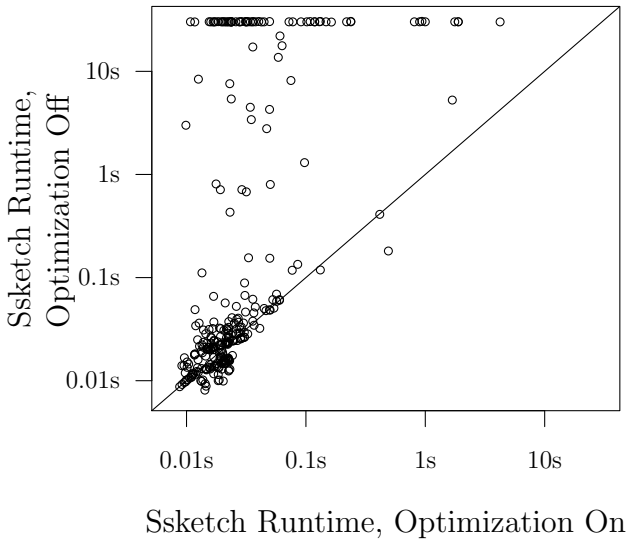

Figure 6-7: Impact of Smten peephole optimizations on Ssketch.

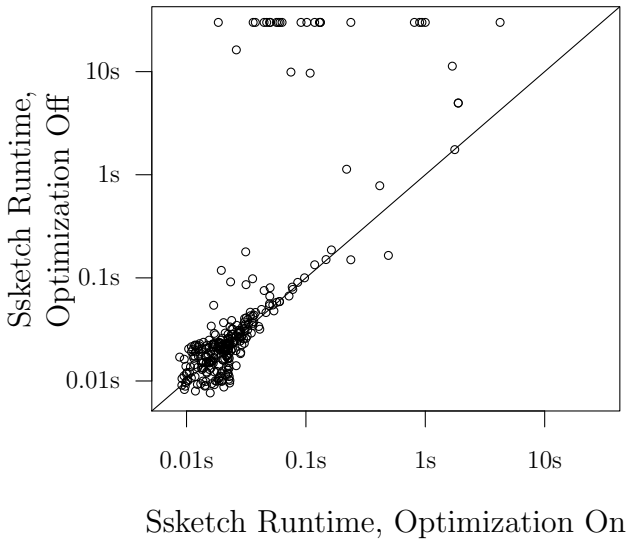

Figure 6-8: Impact of Smten sharing optimization on Ssketch.

affected by the optimizations, be it the peephole optimizations or preservation of sharing.

6.5 Discussion

Purely functional programming languages, such as Haskell, are well suited for describing search computations, because the constraints supported by SAT and SMT solvers must be pure functions. If user-level constraints have side-effects, such as updates to global variables, it is not obvious when those side-effects should occur during search.

Functional programs tend to be more declarative in nature than their corresponding imperative programs. Functional languages put more emphasis on describing the desired result of a computation than the specific strategies used to compute the result. For example, there is no explicit memory allocation or deallocation in a Haskell program, whereas, in C, explicit memory allocation and deallocation is used to describe how to execute a computation. If the runtime system is free to choose the best strategy for executing computation, it can perform well for both standard execution of programs and search-space execution, which requires a different execution strategy.

Nonstrict functional languages are particularly well suited for describing search computations, because they naturally support computations involving infinite data structures, which are more common in search than traditional programming.

One challenge we discovered while using Smten to develop complex applications is that programs are more sensitive to small algorithmic changes than normal Haskell programs. In the construction of queries, Smten conceptually evaluates a function by executing all possible cases of the function. As a consequence, the complexity of all cases plays an important role in determining performance, not just the average case.

The best representation of a function for search in Smten may be different than the best representation for a concrete computation. Attempts to improve performance of the common case can hurt performance when all cases must be considered, because these attempts may increase the number of cases that must be considered.

For example, Figure 6-9 shows two implementations of a list update function that

```
update1 :: Int → a → [a] → [a]
update1 n v [] = []
update1 n v (x:xs) =
  if n == 0 then v:xs
            else (x : update1 (n-1) v xs)

update2 :: Int → a → [a] → [a]
update2 n v [] = []
update2 n v (x:xs) =
  (if n == 0 then v else x ) : update2 (n-1) v xs
```

Figure 6-9: Alternative list update functions in Smten.

exemplify a number of cases we encountered when programming with Smten. The function `update1` uses short-circuiting: when the position in the list where the update should occur has been reached, the tail of the list is returned unmodified. In contrast, `update2` always traverses the entire list.

When we are in a `Space` computation where `n` is unknown but the length of the list is known, `update2` takes time linear in the length of the list to evaluate all possible cases. Each element of the list will be replaced with a ϕ-conditional expression representing whether that element of the list was updated or not. The function `update1`, however, takes time quadratic in the length of the list to evaluate all possible cases, because there is a separate path of computation for each value of `n`, and each of those `n` paths of computation must test on the order of `n` elements to identify the position where the update should occur.

We believe it is not too onerous for a Smten user, whose goal is to simplify their use of SMT, to understand at a high-level what the implications of their algorithmic choices are when conceptually all cases must be evaluated to construct a query.

Chapter 7

Conclusion

7.1 Moving Forward with Smten

This thesis presented Smten, an extension to the Haskell language for combinatorial search problems. We presented an interface to Smten search and discussed what behavior to expect for search spaces involving nonterminating or infinite computation. We showed a general approach to implementing Smten search, and developed a compiler for the Smten language. To evaluate Smten, we used the Smten language to develop tools for three combinatorial search applications: string constraint solving, model checking, and program synthesis. Our findings show that Smten can reduce development effort by orders of magnitude over direct SAT and SMT search.

We believe we have established Smten as a viable approach to developing combinatorial search applications, and as part of this thesis, we have developed a solid baseline implementation of the Smten search approach. Smten is now ready for serious use, which is the next phase of important work to understand more deeply the challenges of developing combinatorial search applications with Smten, and to begin exploring the many opportunities Smten offers for creating and sharing code across combinatorial search applications.

As Smten is adopted by a wider audience, we hope to see the development of libraries of data structures for search. For example, in traditional programming, a hash table, balanced binary tree, or association list may be the most appropriate

choice of data structure, depending on how the structure is used. We believe with Smten there will be a variety of data structures that are suited to different kinds of search, depending on what parts of the data used to access the structures are partially or fully concrete.

We also would like to see, during this phase of Smten use, development of better performance debugging tools. Our experience using Smten for string constraint solving and program synthesis is that the most difficult part of the development process is understanding why a query uses up all the memory on the machine. Once the cause is identified, understanding and implementing a fix for the problem is easy. We can simplify debugging of the performance problems by providing tools for tracing known facts about partially concrete expressions and making those facts available to the developer.

Aside from using Smten for serious applications, we will discuss different categories of experimental work that would be interesting to do on Smten in the future.

7.1.1 Low Hanging Performance Opportunities

Upgrade to GHC 7.8

The implementation of the Smten compiler is based on GHC version 7.6. There is a bug in GHC version 7.6 that, along with the current approach to implementing Smten as a plugin, requires we disable optimizations in GHC in the first phase of the Smten compiler.

Recently GHC version 7.8 has been released, which potentially fixes this limitation. It should hopefully be straight-forward to upgrade the Smten compiler to use GHC version 7.8 and re-enable GHC optimizations. We expect this to improve Smten performance by a small but noticeable factor, perhaps 10 to 50 percent for all applications.

At the time of writing this thesis, the latest release of GHC 7.8 had a bug in the StableName extension, which Smten uses for simplifying SAT and SMT formulas. After GHC 7.8 stabilizes, upgrading to GHC 7.8 would be a good opportunity to

improve the performance of Smten.

Improve Solver Interfaces

Most of our efforts for interfacing with solvers have been focused on the Yices2 solver. An easy way to improve Smten performance would be to devote more effort to cleaning up the rest of the solver interfaces.

7.1.2 Opportunities for Interesting Experimentation

Implied Value Concretization

One of the most interesting opportunities for improving Smten search is using a technique called *implied value concretization* [8].

The idea behind implied value concretization is that in some contexts, the values of free variables are implied directly. For example, consider the following expression:

```
do x ← free_Integer
   let z = if (x == 5)
               then factorial x
               else x * x
   guard (z ≥ 5 && z ≤ 10)
```

In the `then` branch of the conditional expression, the value of x is implicitly 5, because otherwise the `then` branch would not be taken. We would like to optimize the query to:

```
do x ← free_Integer
   let z = if (x == 5)
               then factorial 5
               else x * x
   guard (z ≥ 5 && z ≤ 10)
```

Now, instead of calling factorial on a partially concrete value, the argument is entirely concrete, which means the factorial can be computed normally instead of

inlining it into the generated SAT or SMT query.

Static analysis of the code can reveal some opportunities for implied value concretization, but many of the opportunities for implied value concretization must be discovered dynamically at runtime.

By inspection of the queries Smten currently generates, it is clear there are many opportunities for using implied value concretization to drastically simplify the SMT queries generated. The reason Smten does not already perform implied value concretization is because implied value concretization is at direct odds with preservation of sharing. In one case, expressions are specialized for each context in which they appear, in the other, the goal is to share as much as possible the same expression in different contexts.

For example, consider the search space:

```
do x ← free_Integer
   y ← free_Integer
   let w = y * y * x
   let z = if (x == 5)
               then factorial w
               else w
   guard (z ≥ 5 && z ≤ 10)
```

In this example, the then branch can be partially specialized by replacing `y * y * x` with `y * 5 * y` in the argument to `factorial`:

```
do x ← free_Integer
   y ← free_Integer
   let w = y * x * y
   let z = if (x == 5)
               then factorial (y*5*y)
               else w
   guard (z ≥ 5 && z ≤ 10)
```

But that means parts of the expression `y * x * y` are duplicated rather than

shared. It is not obvious if this will lead to an overall performance improvement. There are also practical challenges to support implied value concretization with preservation of sharing in an efficient manner in the Smten runtime.

Improved Heuristics for Nontermination

The current implementation of Smten uses primitive heuristics for when to approximate a sub-term of a formula. Improving the heuristics could enable a larger class of programs to work out of the box, without deep understanding from the developer to ensure their queries are properly bounded. Good heuristics could also improve performance of search by aggressively pruning paths, leading to simpler SAT and SMT queries.

More Advanced Solvers

Currently, the implementation of Smten supports five primitive solvers and a debug solver combinator. Additional solver combinators could be provided that allow the user to describe more complex and interesting solvers. For example, a generic portfolio solver combinator could be added to Smten that executes the query using multiple solvers in parallel and returns the first result available [2].

7.1.3 Future Language-Level Research

Allow Nested Searches

The current search interface disallows nested searches: queries about queries. This is because nested searches would lead to quantifier alternation in the generated query, which is not well supported by SAT and SMT solvers.

From the user's perspective, nested searches are a powerful feature, which can be used to describe many interesting searches. For example, with nested searches, the program synthesis question can be asked directly using a single Smten search query. If nested searches can be supported efficiently, this would be a nice feature to add to the Smten search interface.

Support for SAT Extensions

There are extensions for SAT, such as MAX-SAT, which find the assignment satisfying the most clauses of the SAT formula. It would be interesting to explore how these extensions could be exposed in the Smten search interface and whether it provides any additional power to the user or opportunities for performance improvement.

Reflection of Partially Concrete Expressions

One question that has presented itself a few times during the development of Smten is whether it would be good to expose more about the underlying representation of a partially concrete object to the user. By exposing the structure of the underlying representation, a user with expert knowledge may be able to provide domain-specific rewrite rules to improve the search space. Evaluating whether this feature would provide any benefits and what the consequences it has on modularity would make for interesting research.

7.2 Closing Remarks

We have presented Smten, a nonstrict, functional programming language for expressing satisfiability-based search problems. Search problems in Smten are specified by directly describing constraints on the search space. The Smten runtime system lazily constructs and executes SMT queries to perform the search on behalf of the user.

Smten drastically simplifies the task of developing complex, high-performance, satisfiability-based search applications by automating the critical but tedious start-up costs and optimizations required to effectively use SMT solvers, allowing the developer to describe the search problem directly, decoupling important design decisions from query construction, easing exploration of the design space, and enabling reuse in the form of libraries that work well in large areas of the design space.

Our evaluation of Smten on substantial satisfiability-based search applications clearly demonstrates that applications developed using Smten require significantly fewer lines of code and less developer effort for performance comparable to stan-

dard SMT-based tools. We hope a consequence of Smten will be that more high-performance satisfiability-based search applications are developed, especially in specialized domains that previously could not justify investing the high costs required to develop an efficient SMT-based tool. We also hope to see a collaborative effort to produce high quality library codes reused across a wide range of different satisfiability-based search applications.

Bibliography

[1] Siddharth Agarwal. Functional SMT solving: A new interface for programmers. Master's thesis, Indian Institute of Technology Kanpur, June 2012.

[2] Martin Aigner, Armin Biere, Christoph Kirsch, Aina Niemetz, and Mathias Preiner. Analysis of portfolio-style parallel SAT solving on current multi-core architectures. In Daniel Le Berre, editor, *POS-13*, volume 29 of *EPiC Series*, pages 28–40, 2014.

[3] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 825–886. IOS Press, 2009.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org`, December 2010.

[5] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 1999.

[6] Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *J. ACM*, 41(4):630–647, July 1994.

[7] G. Cabodi, P. Camurati, and M. Murciano. Automated abstraction by incremental refinement in interpolant-based model checking. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '08, pages 129–136, Piscataway, NJ, USA, 2008. IEEE Press.

[8] 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.

[9] Benjamin Chambers, Panagiotis Manolios, and Daron Vroon. Faster SAT solving with better CNF generation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1590–1595, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[10] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. *CoRR*, abs/1208.2925, 2012.

[11] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14, New York, NY, USA, 2013. ACM.

[12] W F. Clocksin and Christopher S. Mellish. *Programming in Prolog (2nd Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[13] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[14] N. Dave, M. Katelman, M. King, Arvind, and J. Meseguer. Verification of microarchitectural refinements in rule-based systems. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 61–71, July 2011.

[15] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[16] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[17] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008.

[18] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. 2006.

[19] Levent Erkok. http://hackage.haskell.org/package/sbv-3.0, 2014.

[20] Matthew Flatt. Creating languages in Racket. *Commun. ACM*, 55(1):48–56, January 2012.

[21] John Gallant, David Maier, and James Astorer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980.

[22] Vijay Ganesh and David Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer Berlin / Heidelberg, 2007.

[23] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[24] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

[25] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[26] John Hughes and John ODonnell. Expressing and reasoning about non-deterministic functional programs. In Kei Davis and John Hughes, editors, *Functional Programming*, Workshops in Computing, pages 308–328. Springer London, 1990.

[27] William N. N. Hung, Xiaoyu Song, Guowu Yang, Jin Yang, and Marek Perkowski. Quantum logic synthesis by symbolic reachability analysis. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 838–841, New York, NY, USA, 2004. ACM.

[28] 2010 hardware model checking competition. `http://fmv.jku.at/hwmcc10/` , 2010.

[29] INRIA. *The Coq Proof Assistant Reference Manual*, August 2012.

[30] David J. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, Boston, MA, USA, 1996.

[31] 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.

[32] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 192–203, New York, NY, USA, 2005. ACM.

[33] Ali Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of Z3: Integrating SMT and programming. In Nikolaj Bjrner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 400–406. Springer Berlin / Heidelberg, 2011.

[34] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 151–164, New York, NY, USA, 2012. ACM.

[35] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. *SIGPLAN Not.*, 45(6):316–329, June 2010.

[36] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM.

[37] K.RustanM. Leino. Dafny: An automatic program verifier for functional correctness. In EdmundM. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer Berlin Heidelberg, 2010.

[38] Simon Marlow and Others. Haskell 2010 language report. http://www.haskell.org/onlinereport/haskell2010, April 2010.

[39] Yu.V. Matiyasevich. Diophantine representation of enumerable predicates. *Mathematical notes of the Academy of Sciences of the USSR*, 12(1):501–504, 1972.

[40] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, volume 26 of *Studies in Logic and the Foundations of Mathematics*, pages 33–70. Elsevier, 1959.

[41] K.L. McMillan. Interpolation and SAT-based model checking. In Jr. Hunt, WarrenA. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2003.

[42] Alan Mishchenko, Robert Brayton, Jie-Hong R. Jiang, and Stephen Jang. Scalable don't-care-based logic optimization and resynthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):34:1–34:23, December 2011.

[43] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[44] Gopalan Nadathur and Dale Miller. An overview of Lambda Prolog. Technical report, Durham, NC, USA, 1988.

[45] Simon Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stable names in haskell. In Pieter Koopman and Chris Clack, editors, *Implementation of Functional Languages*, volume 1868 of *Lecture Notes in Computer Science*, pages 37–58. Springer Berlin Heidelberg, 2000.

[46] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *J. Funct. Program.*, 19(6):663–697, November 2009.

[47] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM.

[48] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, London, UK, UK, 2000. Springer-Verlag.

[49] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008. AAI3353225.

[50] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 404–415, New York, NY, USA, 2006. ACM.

[51] Rok Sosic and Jun Gu. A polynomial time algorithm for the n-queens problem. *SIGART Bull.*, 1(3):7–11, October 1990.

[52] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[53] Don Stewart. http://hackage.haskell.org/package/yices-painless-0.1.2, January 2011.

[54] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 298–315, Berlin, Heidelberg, 2011. Springer-Verlag.

[55] Jorma Tarhio and Esko Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57(1):131–145, 1988.

[56] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM.

[57] G.S. Tseitin. On the complexity of derivation in propositional calculus. In JrgH. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.

[58] Richard Uhler and Nirav Dave. Smten: Automatic translation of high-level symbolic computations into SMT queries. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 678–683. Springer Berlin Heidelberg, 2013.

[59] Richard Uhler and Nirav Dave. Smten with satisfiability-based search. In *OOPSLA*, OOPSLA '14. ACM, 2014.

[60] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. HALO: Haskell to logic through denotational semantics. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 431–442, New York, NY, USA, 2013. ACM.

[61] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 12 1992.

[62] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[63] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In *OOPSLA*, pages 65–82, 2011.

[64] http://yices.csl.sri.com/index.shtml, August 2012.

[65] Ramin Zabih, David McAllester, and David Chapman. Non-deterministic lisp with dependency-directed backtracking. In *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 59–64. AAAI Press, 1987.