Smten with Satisfiability-Based Search

Richard Uhler
MIT-CSAIL
ruhler@csail.mit.edu

Nirav Dave
SRI International
ndave@csl.sri.com

Abstract

Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) have been used in solving a wide variety of important and challenging problems, including automatic test generation, model checking, and program synthesis. For these applications to scale to larger problem instances, 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. We present Smten, a high-level language for orchestrating and constructing satisfiability-based search queries. We show that applications developed using Smten require significantly fewer lines of code and less developer effort to achieve results comparable to standard SMT-based tools.

Keywords Satisfiability, Satisfiability Modulo Theories, Metaprogramming, Domain-Specific Language, Haskell

1. Introduction

Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) have been leveraged in solving a wide variety of important and challenging problems, including automatic test generation [5], model checking [3, 28, 31], and program synthesis [33].

Problems that benefit the most from SAT and SMT are often NP-hard combinatorial search problems that face significant challenges when scaling to larger problem instances. Scaling is important, because whether these applications can be applied to practical problems depends largely on how well they scale to larger problem instances.

In principal, using a SAT/SMT solver allows a designer to sidestep developing a sophisticated custom search technique for their problem, reducing the time and effort required to develop a solution that performs well. However, in practice the construction of the SAT/SMT queries can significantly impact performance, and designers are forced to exert non-trivial effort to optimize query generation. What may have been a relatively natural translation from a high-level problem to a SAT/SMT query in reality requires development of a large custom computational framework for constructing and interpreting an optimized query, requiring many man-months of effort to implement and debug.

To illustrate this issue more concretely, consider the application of string-constraint solving exemplified by the SMT-based tool HAMPI [20]. In HAMPI the goal is to synthesize a bounded-size string based on a template that belongs to the language of a given context-free grammar or regular expression. For instance, given the template `ab??cd`, where `?` stands for an arbitrary character, and given the regular expression `y(z*(ab)*)* | (((ab)*(cd)*)*)`, the tool may synthesize either the string `ababcd` or `abcdcd`. Intuitively, HAMPI constructs an SMT query by unrolling the regular expression match computation over the symbolic string input. However, to be practical, query construction must avoid unrolling obviously false paths (in our example, the first alternative can never match, as every string from the template starts with `a`, not `y`), and capture sharing (in our example, the substring match for `cd`, the last two characters in the string template, is the same whether the outermost Kleene star in the second alternative is repeated once or twice). These optimizations improve performance by orders of magnitude, but require thousands of lines of Java to implement.

Not only does a large code base take significant time to produce, it is also more costly to modify, making it harder to refine the tool and limiting design exploration. For instance, much of the reason HAMPI is efficient is because it exploits the known size of the target string to simplify the regular expression. A natural extension is to allow string templates to represent variable-size strings. To synthesize a variable-size target string, the designer has two obvious strategies: he can make a specialized query for each possible size and exploit size-level optimizations, or construct a single query encoding all string sizes that exposes sharing for different sizes to the SMT solver. It is not obvious a priori which approach will offer the best performance; a developer would want to implement both approaches to evaluate their relative

merit. However, given our initial HAMPI implementation, it is straightforward to implement only the first approach; the second approach requires significant modification to the framework's handling of string templates and integration of a richer set of size-based optimizations into the already large framework.

In essence, while SMT has succeeded in allowing designers to share computational cores across many disparate tools, it does not simplify the tasks of query orchestration and generation necessary to leverage the shared computational core of SMT effectively. As a result, despite providing significant value, SMT technologies are used by only those with tasks that merit enormous amounts of developer effort and who have sufficient development experience to build a framework of sufficient complexity: a small set of experts.

Our goal is to reduce the overhead of developing SMT-based tools and enable non-experts to enjoy the development advantages that SMT provides. To this end, we have developed Smten, a high-level language for orchestrating and constructing satisfiability-based search queries. Smten is based on the functional programming language Haskell [27] and allows a developer to concisely specify the orchestration of search queries, while automatically optimizing query construction. The designer describes the high-level decomposition of their task into SMT queries directly, resulting in significant improvements to code size, code clarity, and development time. Smten also provides value to expert SMT developers in the form of better support for modularity, portability, and reuse.

The specific contributions of this paper are:

- We present the Smten programming abstraction for high-level orchestration of search problems, inspired by Haskell's list monad and amenable to automatic reduction to the problem of satisfiability (Section 3).
- We provide precise semantics for Smten's search (Section 4).
- We describe a general framework for optimized construction of SAT queries for search problems expressed using the Smten programming abstraction (Section 5).
- We describe an abstraction-refinement based approach for supporting search descriptions involving unbounded computation (Section 5.1).
- We identify key optimizations in Smten crucial for performance in practice that benefit a wide range of applications (Section 6).
- We have built a compiler for Smten based on the Glasgow Haskell Compiler [15], and have used it to develop substantial applications (Section 7).
- We demonstrate experimentally via our applications from the domains of constraint solving, model checking, and program synthesis that Smten enables (Section 8):
  - More intuitive and concise implementations
  - Ease of design exploration and algorithmic refinement, leading to improved performance
  - Performance comparable to existing state-of-the-art hand-optimized implementations based on SAT and SMT solvers
- We compare our approach to related work (Section 9).
- We discuss the benefits of a non-strict language for search, report on our experience using Smten to develop applications, and present relevant topics for future research (Section 10).

		v	$\in$	*Boolean Variable*
Boolean		b	::=	`true` | `false`
Formula		ϕ	::=	$b \mid v \mid \neg\phi$
			|	$\phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$
Abbr	`ite` $\phi\ \phi_1\ \phi_2$		=	$(\phi \wedge \phi_1) \vee (\neg\phi \wedge \phi_2)$

Figure 1. Syntax of a boolean formula.

2. Background

Automatic test generation, model checking, and program synthesis belong to a class of applications we call *satisfiability-based search* applications. The high-level goal in each of these applications is to determine whether there exists any solution to the problem, and if so, to find a single instance of the potentially many solutions to the problem. For example, automatic test generation searches for program inputs that exercise a particular path in a program; model checking searches for counter-examples to desired properties in models of hardware or software systems; and program synthesis searches for a program satisfying a given specification.

A general approach used in state-of-the-art tools for solving satisfiability-based search applications is to reduce the search problem to the problem of satisfiability and leverage a SAT or SMT solver to efficiently solve it. An entirely different approach to solving satisfiability-based search applications is based on non-strict evaluation and the list monad in Haskell, which provides a direct, modular, and concise way to describe search problems, but suffers from combinatorial explosion. The Smten language combines these two approaches, providing a direct, modular, and concise way to describe search problems that are automatically reduced to the problem of satisfiability and solved efficiently with the help of SAT and SMT solvers.

2.1 Satisfiability

The problem of Satisfiability is to determine if there exists an assignment to variables of a given boolean formula under which the formula is satisfied. The syntax for a boolean formula is shown in Fig. 1. We use the abbreviation `ite` $\phi\ \phi_1\ \phi_2$ for a commonly occurring if-then-else construct. We will often use μ as a variable representing a satisfying assignment

to a boolean formula. An assignment μ is a mapping from variable to boolean value.

A SAT solver takes a boolean formula as input and returns a satisfying assignment for the formula if one exists, or otherwise indicates the formula is unsatisfiable. Specifically, given boolean formula ϕ, a SAT solver returns `Sat` μ if the formula is satisfiable with μ an arbitrary satisfying assignment, and `Unsat` if the formula is unsatisfiable.

Though the problem of satisfiability is NP-complete [8], SAT solvers leverage efficient heuristic search algorithms based on DPLL [9, 10] for satisfiability that scale well for many practical problem instances due to decades of research and implementation efforts.

Satisfiability Modulo Theories (SMT) is an extension to the problem of Satisfiability with additional background theories for which there exist decision procedures [2]. For instance, an SMT solver may support formulas with integer variables and linear arithmetic operations, or bit-vector variables with bit-vector operations. SMT solvers leverage higher-level information when solving the formula, *e.g.,* the commutativity and associativity of the operator `+`.

2.2 Reducing Search to Satisfiability

Consider the string-constraint solving example from the introduction as an example of how a satisfiability-based search problem can be reduced to an SMT formula. An SMT query can be constructed for the problem of synthesizing a string from the template `ab??cd` matching the regular expression `(ab)*(cd)*`. At a high level, the desired query is:

$$(\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"ababab"}\ ) \vee (\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"ababcd"}\ ) \vee$$
$$(\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"abcdcd"}\ ) \vee (\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"cdcdcd"}\ )$$

Here the free variables x_1 and x_2 have been used to represent the value of the unknown characters. The solver is responsible for finding values of x_1 and x_2 that satisfy the formula.

Unfortunately, SMT solvers do not support these high-level operations on character strings. Instead, the high-level query on character strings can be encoded using a background theory supported by the solvers, such as a theory of bit vectors. Using ASCII to encode characters as bit vectors, the characters `a`, `b`, `c`, and `d` map to bit-vector values `97`, `98`, `99`, and `100` respectively; the free variables x_1 and x_2 are treated as bit-vector variables; equality of strings is broken down into element wise character equality; and equality of characters is mapped to equality of bit vectors. With this encoding, we arrive at the following SMT query:

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

Under the assumption that the number of characters in the string is fixed, it is relatively straightforward to automate the construction of this query by implementing a regular-expression match algorithm that operates on strings of encoded characters instead of strings of normal characters. The result of running the regular-expression match will be an SMT encoding of a boolean where the equality operator, logical AND, and logical OR construct a boolean formula instead of evaluating to a boolean value.

Note that this query can be simplified before being sent to the solver, resulting in the simpler query:

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

If the number of characters in the string is not fixed, this encoding does not work. It is not immediately obvious what encoding to use in this case. Regardless, the regular-expression match algorithm will have to change, because it must operate on an encoding of a string of encoded characters instead of a direct list of encoded characters. If there is a non-zero lower bound on the number of characters in the string, it may make sense to implement a hybrid of these two approaches to use the most efficient encoding of the regular-expression match for both the fixed-size prefix of the string and the variable-size suffix.

2.3 SMT User Challenges

Aside from the challenges of encoding high-level queries as SMT queries and optimizing the construction of the SMT queries, there is a fair amount of tedious work required to use an SMT solver. This includes deciding which solver to use, installing the solver, understanding whether to use a text-based or native interface to the solver, learning the syntax of queries or the proper sequence of API calls to pose the query and interpret the model, knowing how to correctly configure the solver, and understanding memory management requirements for the solver.

For users who have limited or no knowledge of SMT, the startup costs of using SMT can be prohibitively high. Expert users of SMT with enough incentive to overcome the tedium of using SMT solvers face the problem that their tools are optimized for specific solvers, encodings, and query orchestrations, making it impractical to reuse code for different applications and domains.

2.4 Modular Search with Haskell's List Monad

An entirely different approach to solving satisfiability-based search applications is based on non-strict evaluation and the list monad in Haskell. The approach is to enumerate all possible candidate solutions to the search computation and filter out invalid cases one-by-one until a satisfactory solution is found: in essence, a simple chronological backtracking search.

The value added by the functional programming language Haskell is its non-strict evaluation semantics, and special `do`-notation for monadic programming that can be used with the

```
match :: RegEx → String → Bool
match = ...

candidates :: [String]
candidates = do
  x1 ← ['a'..'z']
  x2 ← ['a'..'z']
  return ['a', 'b', x1, x2, 'c', 'd']

solutions :: [String]
solutions = do
  candidate ← candidates
  guard (match "(ab)*(cd)*" candidate)
  return candidate

main :: IO ()
main = case solutions of
    [] → putStrLn "No Solution"
    (x:_) → putStrLn ("Solution: " ++ x)
```

Figure 2. Haskell list monad approach to searching for a string from `ab??cd` matching `(ab)*(cd)*`.

list type. Non-strict semantics mean functions are evaluated only when they are needed for the result of the computation. As a consequence, in Haskell a candidate solution need not be checked in its entirety before discovering it is an invalid solution, and evaluation stops as soon as the first valid solution is found, if only the first solution is inspected. Coupled with Haskell's `do`-notation and standard operations on the list type, non-strict evaluation provides an elegant way to describe search problems.

For example, Fig. 2 shows a sketch of Haskell code that could be used to search for a string from the template `ab??cd` that matches the regular expression `(ab)*(cd)*`.

The function `candidates` produces a list of all concrete strings described by the template in our example. The notation `['a'..'z']` is an arithmetic sequence that produces the list of all characters from `'a'` to `'z'`.

The `do`-notation can be thought of as a set comprehension:

```
{ ['a', 'b', x1, x2, 'c', 'd'] | x1 ∈ ['a'..'z'],
                                 x2 ∈ ['a'..'z'] }
```

Technically `do`-notation desugars into calls of the function `map`, which applies a function to every element of a list, and `concat`, which flattens a list of lists into a single list of elements:

```
concat (map (λx1 →
    concat (map (λx2 →
       [['a', 'b', x1, x2, 'c', 'd']]
       ) ['a'..'z'])
    ) ['a'..'z'])
```

The `solutions` function uses `guard` to filter out those candidate strings that match the given regular expression.

Note from the type signature of the function `match` that its implementation is given for ordinary character strings, even though it is applied to a set of possible strings. The particular set of strings being applied is not built into the implementation of `match` as it was when encoding SMT queries. In this sense, the list monad approach for describing search problems is much more modular than the SMT approach. Of course, the list approach is also terribly inefficient for problems of non-trivial complexity as it fails to recognize many important opportunities for sharing when evaluating the `match` function, resulting in exploding run times.

The idea behind Smten is to express search problems using the more modular list monad approach, then automatically construct SMT queries based on this description rather than re-evaluating functions for each element of a list.

The Smten runtime handles optimization of query construction, and takes care of the tedium of interfacing with the SMT solver. This will lower the barrier to entry for new users with limited or no knowledge of SMT.

Expert SMT users with existing SMT-based tools in different frameworks can also benefit from Smten, but not in as immediate a fashion. Our experiments suggest porting an existing SMT-based tool to Smten requires less effort than the original implementation of the tool, based on orders of magnitude code size reduction, but this effort is still non-trivial. The real benefits of Smten for expert SMT users come after their tools have been ported to Smten. Smten simplifies maintenance of these tools and design exploration for future tool improvements. Most importantly, Smten's modularity makes it feasible to construct and reuse libraries for search across a broad range of domains and applications.

3. Search with Smten

In this section we present an extension to the Haskell language for describing search problems. We refer to the combination of Haskell and this extension as Smten. Our goal is to provide an abstraction with the same descriptive benefits as the list monad, but which is also suited to efficient evaluation using SAT/SMT.

In Smten, instead of representing a search space as a list of candidate solutions, we introduce a primitive search space type analogous to the list type in Haskell. The key differences between Smten search spaces and Haskell's list approach that enable us to perform search intelligently by reduction to SMT queries are:

- We leverage the property of search that we are interested in only a single solution instead of all valid solutions, allowing extraneous computation to be skipped.
- We exploit the fact that we may accept any solution, not just the first possible satisfying solution.
- We disallow nesting searches, as this would require the evaluation of an internal search that SAT solvers do not support.

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

Figure 3. Interpretation of Smten search space primitives.

3.1 The Smten Search Interface

Our extension provides the following abstract data type and operations:

```
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

search :: Space a → IO (Maybe a)
```

Conceptually, an expression of type `Space a` describes a set of elements of type `a`. However, for our purposes, it is helpful to think of expressions of type `Space a` as describing a search space for elements of type `a`, because Smten does not need to construct the entire set to find a single element of it.

Figure 3 shows the meaning of the primitives for describing search spaces along with the corresponding Haskell list operation. The primitive `empty` is the empty search space and `single` e is a search space with a single element e. The primitive `union` s_1 s_2 is the union of two search spaces, and differs from the corresponding list operation s_1 `++` s_2 in that it does not place the restriction that elements in s_1 are searched before elements in s_2. 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. As with `union`, `join` is different from its corresponding list operation `concat` in that it does not specify an order of the elements.

The `search` primitive is used to search a space and is the only new primitive operation not used to describe search spaces. 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 `search` primitive is not a pure function, because it is non-deterministic and may return any possible e. It is standard practice in Haskell to treat non-pure functions like `search` as IO computations to preserve referential transparency. Marking `search` as an IO computation has the added benefit of prohibiting nested searches.

3.2 An Example of Search in Smten

Figure 4 shows a complete implementation of our string constraint solver in Smten.

As is natural in a functional language, regular expressions are represented using an algebraic data type (Line #1). Here we implicitly make use of sum, product, and recursive types.

The `match` function (Line #4) uses pattern matching and higher-order functions to test whether a string matches a regular expression. Note, the description for `match` can be used directly with *concrete* strings, whose length and characters are all fully known.

The string template is described using an algebraic data type `Template` (Line #18). A template is either `Str` for a concrete string, `Cat` for the concatenation of strings formed from two templates, or `Free` for a variable string whose length ranges between the given bounds and whose characters may be any value.

The function `candidates` (Line #32) takes a template and returns a search space computation of all candidate strings defined by that template. The first clause of the `candidates` function handles the `Str` case and produces a singleton search space with the concrete string `s`. The second clause handles the `Cat` case by concatenating the two search spaces defined by the sub-templates `ta` and `tb`. The final clause instantiates a free string via the helper functions `choose` and `replicateM`. The `choose` function (Line #21) converts a list of elements into a search space by recursively constructing a union of singleton spaces of each list element. The function `replicateM` (which is part of the standard library in Haskell) replicates its arguments the given number of times and returns the cross product of all possible results (Line #25). For example, the search space described by the expression `replicateM 2 ['a' .. 'c']` represents the set of 9 strings `"aa"`, `"ab"`, `"ac"`, `"ba"`, `"bb"`, `"bc"`, `"ca"`, `"cb"`, and `"cc"`. Thus, the `do`-notation for the third clause of `candidates` can be interpreted as meaning for each possible sequence `chars` of `h` characters, each character of which is one of the characters from `'a'` to `'z'`, and for each possible width `w` between `l` and `h`, take `w` characters from `chars`, and include that string in the search space.

The function `solutions` (Line #43) takes a template and regular expression and produces the search space of all candidate strings from the template that match the regular expression. It does this by first calling `candidates` to produce the search space of all strings generated by the template, then uses `do`-notation, which desugars into a `join` of a `map`, to map each non-matching string into the empty set and each matching string into a singleton set, and join the results into a single set which contains only those strings that matched the regular expression.

The function `solve` (Line #50) calls `search` to evaluate the search, and prints the solution found, if any.

An important point about this implementation of string constraint solving is that the description of regular expres-

```
1  data RegEx = Empty | Epsilon | Atom Char
2   | Star RegEx | Concat RegEx RegEx | Or RegEx RegEx

4  match :: RegEx → String → Bool
5  match Empty        _  = False
6  match Epsilon      s  = null s
7  match (Atom x)     s  = s == [x]
8  match r@(Star x)   [] = True
9  match r@(Star x)   s  = any (match2 x r)
10                           (splits [1..length s] s)
11 match (Concat a b) s  = any (match2 a b)
12                           (splits [0..length s] s)
13 match (Or a b)     s  = match a s || match b s

15 match2 a b (sa, sb)   = match a sa && match b sb
16 splits ns x           = map (λn → splitAt n x) ns

18 data Template =
19   Str String | Cat Template Template | Free Int Int

21 choose :: [a] → Space a
22 choose [] = empty
23 choose (x:xs) = union (single x) (choose xs)

25 replicateM :: Int → Space a → Space [a]
26 replicateM 0 s = single []
27 replicateM n s = do
28   hd ← s
29   tl ← replicateM (n-1) s
30   single (hd:tl)

32 candidates :: Template → Space String
33 candidates (Str s) = single s
34 candidates (Cat ta tb) = do
35   a ← candidates ta
36   b ← candidates tb
37   single (a ++ b)
38 candidates (Free l h) = do
39   chars ← replicateM h (choose ['a'..'z'])
40   w ← choose [l..h]
41   single (take w chars)

43 solutions :: Template → RegEx → Space String
44 solutions t r = do
45   s ← candidates t
46   if (match r s)
47     then single s
48     else empty

50 solve :: Template → RegEx → IO ()
51 solve t r = do
52   result ← search (solutions t r)
53   case result of
54     Nothing → putStrLn "No Solution"
55     Just x → putStrLn ("Solution: " ++ show x)
```

Figure 4. String constraint solver implemented with Smten.

sions and regular expression matching is direct, reusing standard Haskell code that operates on concrete regular expressions and strings to describe search over a space of strings (and possibly a space of regular expressions, though that is not shown here).

3.3 The Strictness of Search

Smten allows arbitrary expressions in a search space computation. This means the complete construction of a `Space` expression may be non-terminating. Naively one might assume that such a computation would always result in non-termination ($\perp$), but because `search` returns only a single element of the search space, it may be possible to return a result even though the complete search space cannot be constructed. There are many situations where expressing and searching in these necessarily incomplete search spaces is valuable.

For example, in our string constraint solver, we may not wish to put an upper bound on the string length. The following example demonstrates a `Space` expression describing a search space that includes strings of unbounded length:

```
search (union (single "b") (union aStr (single "c")))
 where
  aStr :: Space String
  aStr = union (single "") (map (λs → 'a':s) aStr)
```

In this example, `aStr` is a search space defined recursively that represents the infinite set of strings whose every character is an `'a'`. The top level search happens in a space including the string `"b"`, all the strings from `aStr`, and the string `"c"`. This is a search for elements of the set $\{\texttt{"b"}\} \cup \{\texttt{""}, \texttt{"a"}, \texttt{"aa"}, \ldots\} \cup \{\texttt{"c"}\}$. Operationally, constructing the entire space of strings in `aStr` will never terminate. Because we have not specified an order of elements searched in the space, however, it is not clear whether, for example, `"c"` is a valid result of this search, or if the search may fail to terminate.

Because it can lead to more natural descriptions of problems and better modularity, we chose to take the most non-strict view of search. In this case we allow either `"b"`, `"c"`, or any string from `aStr` to be found, and we require the search terminates eventually.

If there are no valid solutions to a search space for which the complete construction would lead to non-termination, then the entire space must be searched, and the search will fail to terminate. For example, using the `aStr` search space, the following search will fail to terminate:

```
search $ do
  s ← aStr
  if (elem 'b' s)
    then single s
    else empty
```

It is not feasible in general to expect the system could determine none of the infinite set of strings described by `aStr` contain the character `'b'`. If the character `'b'` had been

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

Figure 5. Syntax of KerSmten types and terms.

replaced with the character `'a'` in this example, however, the search would terminate with any one of the strings from `aStr`.

4. Precise Semantics for Smten

In this section we provide a precise semantics for search in Smten to clarify the behavior of search, especially with respect to the strictness of search. These semantics will also serve as a foundation for our presentation of the Smten implementation approach in Sec. 5.

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 non-strict, 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 API.

Figure 5 shows the syntax of types and terms for KerSmten. Precise typing judgments for KerSmten are not included due to space limitations. 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 Sec. 3.1.

For the syntax of terms in Fig. 5, 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, makes it clear what the type of every term is. To reduce clutter, we sometimes omit explicit type annotations when not relevant.

Figure 6 gives a complete small-step structured operational semantics for KerSmten evaluation. The semantics are organized by the three different kinds of computation in Smten: pure evaluation, search, and IO.

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. The transition $e_1 \to_e e_2$ 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.

Search Computation Search-space evaluation reduces terms of type `Space` T to either the empty search space `empty`, or a search space with a single element `single` e. The semantics of search-space evaluation is split into two separate transition relations. The transition $s_1 \to_{s\uparrow} s_2$ describes a search-space *expansion* process, which eliminates top level occurrences of `map` and `join`, resulting in an expanded set of elements described using `empty`, `single`, and `union`. The transition $s_1 \to_{s\downarrow} s_2$ describes a search space *reduction* process, which eliminates top level occurrences of `union` by selecting an arbitrary result.

Search space expansion is type-sound and deterministic. Search space reduction is type-sound, but is *not* deterministic, because it can arbitrarily select a result from either the first or second argument to `union`. Combined, search space expansion and reduction have the effect of freely exploring the search space and selecting a result when one is found.

The rules *sts-union-left* and *sts-union-right* allow part of a search space to be ignored (the space s in these rules), so long as a result has already been found. This is what makes it possible to return a search result when parts of the search space are not completely computable. Because it is possible to ignore parts of a search space only when a result has been found, the entire space must be searched to determine it is empty.

In Sec. 3.3 we said search in a space with some element must terminate eventually. To express this using the precise semantics, we need an additional fairness constraint on application of the rules *sts-union-a1* and *sts-union-a2* not shown here.

IO Computation IO computation applies to terms of type `IO` T and is where search computations are executed. The

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

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

st-fst-a $\dfrac{e \to_e e'}{\texttt{fst}\ e \to_e \texttt{fst}\ e'}$ *st-snd-a* $\dfrac{e \to_e e'}{\texttt{snd}\ e \to_e \texttt{snd}\ e'}$ *st-app-a* $\dfrac{f \to_e f'}{f\ e \to_e f'\ e}$

Search Space Expansion

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

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

sts-pure $\dfrac{s \to_e s'}{s \to_{s\uparrow} s'}$ *sts-map-a* $\dfrac{s \to_{s\uparrow} s'}{\texttt{map}\ f\ s \to_{s\uparrow} \texttt{map}\ f\ s'}$ *sts-join-a* $\dfrac{s \to_{s\uparrow} s'}{\texttt{join}\ s \to_{s\uparrow} \texttt{join}\ s'}$

Search Space Reduction

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

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

sts-s1 $\dfrac{s \to_{s\uparrow} s'}{s \to_{s\downarrow} s'}$

sts-union-a1 $\dfrac{s_1 \to_{s\downarrow} s_1'}{\texttt{union}\ s_1\ s_2 \to_{s\downarrow} \texttt{union}\ s_1'\ s_2}$ *sts-union-a2* $\dfrac{s_2 \to_{s\downarrow} s_2'}{\texttt{union}\ s_1\ s_2 \to_{s\downarrow} \texttt{union}\ s_1\ s_2'}$

IO Computation

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

stio-bind-return	$(\texttt{return}_{io}\ e_1)\ \texttt{>>=}_{io}\ e_2$	$\to_{io}$	$e_2\ e_1$
stio-search-empty	$\texttt{search}\ \texttt{empty}$	$\to_{io}$	$\texttt{return}_{io}\ \texttt{Nothing}$
stio-search-single	$\texttt{search}\ (\texttt{single}\ e)$	$\to_{io}$	$\texttt{return}_{io}\ (\texttt{Just}\ e)$

stio-bind-a $\dfrac{e_1 \to_{io} e_1'}{e_1\ \texttt{>>=}_{io}\ e_2 \to_{io} e_1'\ \texttt{>>=}_{io}\ e_2}$ *stio-pure* $\dfrac{e \to_e e'}{e \to_{io} e'}$ *stio-search-a* $\dfrac{s \to_{s\downarrow} s'}{\texttt{search}\ s \to_{io} \texttt{search}\ s'}$

Figure 6. Semantics of KerSmten.

$$e, f, s \quad ::= \quad x \mid \lambda x_T \,.\, e \mid f\ e \mid \texttt{unit} \mid \cdots \\ \mid \quad \phi\ ?\ e_1\ :\ e_2 \mid \{e \mid \phi\}$$

Figure 7. Augmented syntax of KerSmten terms.

transition $e_1 \rightarrow_{io} e_2$ represents a single step of IO computation. IO computation is type-sound.

The rule *stio-search-empty* applies for a search space that evaluates to `empty`, and *stio-search-single* applies for a search space containing at least one element.

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.

5. Smten Search by SAT

In this section we present a syntax-directed approach for constructing a SAT formula from a Smten description of a search problem.

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 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 Fig. 7. The syntax of boolean formulas ϕ was given in Fig. 1.

ϕ-Conditional Expression $(\phi\ ?\ e_1\ :\ e_2)$ 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 $\phi\ ?\ e_1\ :\ e_2$ 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.

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

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 $\mu_1 = \{(v, \texttt{true}), (w, \texttt{true})\}$ 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:

$$\{e \mid \phi\} = \phi\ ?\ \texttt{single}\ e\ :\ \texttt{empty}$$

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

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 then implemented via a SAT solver as follows:

1. Construct Evaluate the expression s resulting in the construction of a set expression $\{e \mid \phi\}$.

2. Solve 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 some assignment μ, and we return the result `Just` $e[\mu]$, because $e[\mu]$ belongs to the set s.

More precisely, we augment pure evaluation to work in the presence of ϕ-conditional and set expressions, and we define a new evaluation strategy for search space computations that reduces them to set expressions.

Augmenting Pure Evaluation Figure 8 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 reason duplicating primitive operations is not as severe as the duplication of functional calls when using the list monad is because functions whose control flow is independent of their arguments can be executed once, instead of once for every argument.

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

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

Figure 8. KerSmten pure evaluation with new expressions.

For example, the expression $(\lambda\, x \,.\, (x, x))(\phi\, ?\, e_1 \,:\, e_2)$ reduces with standard beta substitution (*st-beta*) to:

$$(\phi\, ?\, e_1 \,:\, e_2,\ \phi\, ?\, e_1 \,:\, e_2)$$

The approach of the list monad 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 list monad approach, but just once using our approach. This is the key idea behind why Smten performs better than the list monad in Haskell.

We discuss in Sec. 6.1 how we can canonicalize partially concrete expressions to avoid duplication in the primitive operations as well.

Search-Space Evaluation Figure 9 gives a new set of rules for evaluating search-space expressions to set expressions. Note that the rules here do not properly describe the strictness properties we need for search. 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 preserve strictness of search.

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.

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:

$$\{\bot \mid \texttt{false}\} = \texttt{false}\, ?\, \texttt{single}\, \bot \,:\, \texttt{empty}$$

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 Sec. 6.2).

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:

$$\{e \mid \texttt{true}\} = \texttt{true}\, ?\, \texttt{single}\, e \,:\, \texttt{empty}$$

Note that the argument e to `single` does not need to be evaluated to put the expression `single` e in 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 $(\texttt{union}\ \{e_1 \mid \phi_1\}\ \{e_2 \mid \phi_2\})$ 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 $v = \texttt{true}$ corresponds to choosing an element from the first set, and $v = \texttt{false}$ 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 $v = \texttt{true}$ 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 the `ite` v `true` `false` is with $v = \texttt{true}$, so the value 1 must be selected.

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

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

$$
\textit{sx-union-a1}\ \frac{s_1 \to_{sx} s_1'}{\texttt{union}\ s_1\ s_2 \to_{sx} \texttt{union}\ s_1'\ s_2} \qquad \textit{sx-union-a2}\ \frac{s_2 \to_{sx} s_2'}{\texttt{union}\ s_1\ s_2 \to_{sx} \texttt{union}\ s_1\ s_2'}
$$

$$
\textit{sx-phi-a1}\ \frac{s_1 \to_{sx} s_1'}{\phi\ ?\ s_1\ :\ s_2 \to_{sx} \phi\ ?\ s_1'\ :\ s_2} \qquad \textit{sx-phi-a2}\ \frac{s_2 \to_{sx} s_2'}{\phi\ ?\ s_1\ :\ s_2 \to_{sx} \phi\ ?\ s_1\ :\ s_2'}
$$

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

Figure 9. SAT-Based search-space evaluation.

sx-map The expression $\texttt{map}\ f\ \{e \mid \phi\}$ 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 $\texttt{join}\ \{s \mid \phi\}$ reduces by the rule *sx-join* to $\phi\ ?\ s\ :\ \texttt{empty}$. This reduction is easy to understand interpreting the set expression $\{s \mid \phi\}$ as the canonical form $\phi\ ?\ \texttt{single}\ s\ :\ \texttt{empty}$. 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 further reduced before reaching 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:

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

This can be compressed to the form:

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

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

5.1 Avoiding Spurious Non-Termination

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. It is somewhat surprising that these rules can result in non-termination for some search spaces that are completely finite. Consider the following search space:

```
search $ do
  x ← union (single True) (single False)
  if (x || not x)
     then single x
     else let y = y in y
```

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

This search is entirely finite. The list monad can produce the exhaustive list of both solutions: `True` and `False`. In our implementation, this evaluates to an expression such as:

$$(x \vee \neg x)\ ?\ (\texttt{single}\ (x\ ?\ \texttt{True}\ :\ \texttt{False}))\ :\ (\texttt{fix}\ (\lambda y.y))$$

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 `true`, 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 second 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. There are two downsides to this approach:

- The SAT solver is called for every condition, which we expect to be prohibitively expensive
- This approach does not help any with searches involving reachable non-terminating paths. An implementation using this approach would be overly strict in evaluation of `search` as described in Sec. 3.3

We take an alternative approach that heuristically detects long running computations and uses 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 results in an implementation of `search` with our desired strictness properties.

In our implementation, we evaluate set expressions non-strictly (contrary to the rules presented in Fig. 9); set expressions are forced only when we need to compute the formula part of the set expression. A consequence of this is we have to concern ourselves only with potential non-termination when looking at the formula being constructed, and not in the constructions for set expressions.

Our goal, then, is to solve the problem of satisfiability for a boolean formula where sub-terms may not be fully computable. The high-level approach is to search first for satisfying assignments to the formula that are not affected by the non-computable parts of the formula.

For example, consider the boolean formula:

$$\phi = \texttt{ite}\ x_0\ \hat{\phi}\ x_1$$

where x_0 and x_1 are boolean variables and $\hat{\phi}$ represents a part of the formula that appears to be uncomputable. Even without knowing the value of formula $\hat{\phi}$, it is possible to find a satisfying assignment for ϕ: take x_0 to be `false` and x_1 to be `true`.

Our runtime heuristically detects sub-terms of formulas that are long-running computations. Rather than compute their values, the runtime treats these sub-terms as black boxes. In the previous example, $\hat{\phi}$ is an example of a black-box sub-term. Given a formula ϕ annotated with black box sub-terms, we generate three new formulas: p, a, and b.

The formula p corresponds to the condition under which the black boxes of ϕ do not affect its value. In the above example, $p = \neg x_0$, because if x_0 is `false`, the value of $\hat{\phi}$ is irrelevant.

The formula a is the same as formula ϕ under the assumption all the black box sub-terms are irrelevant. The black box sub-terms can be replaced with any value, including `true`, `false`, or any other value leading to a simple boolean formula for a. In the above example, if $\hat{\phi}$ is unreachable in ϕ, then $\phi = x_1$, so we take $a = x_1$.

The formula b contains the black box sub-terms and is equivalent to ϕ for all assignments where p is `false`. In the above example, $b = \hat{\phi}$.

The meaning of each of these terms can be summarized with the logical equality $\phi = \texttt{ite}\ p\ a\ b$. In other words, a is a finite approximation to ϕ, the approximation is exact when p holds, and b is used to refine the approximation.

The following abstraction-refinement semi-decision procedure makes use of this construction to progressively refine the black box parts of a boolean formula ϕ until a solution is found. If there exists a solution to the search, it will be found eventually. The procedure is:

1. Construct the terms p, a, and b for ϕ.
2. Check if $p \wedge a$ is satisfiable. If so, the assignment is also a satisfying assignment for ϕ and we are done.
3. Otherwise, check if $\neg p$ is satisfiable. If not, then ϕ is unsatisfiable and we are done. In this case, the formula b is unreachable, so none of the black box sub-terms need to be computed.
4. Otherwise, b may or may not have a solution and we refine our abstraction, but may restrict it with the knowledge that the cases we have considered previously may be ruled out. That is, we repeat this procedure with $\neg p \wedge b$.

6. Optimizations

Many of the optimizations required to make a tool developed with Smten work well in practice are specific to the tool and can be expressed in the user's code. There are a handful of important optimizations, however, which 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 effect 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 Sec. 8.4.

6.1 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:

$$\begin{aligned} & \phi \,?\, (e_{11}, e_{12}) \;:\; (e_{21}, e_{22}) \\ \rightarrow \quad & (\phi \,?\, e_{11} \;:\; e_{21} \;,\; \phi \,?\, e_{12} \;:\; e_{22}) \end{aligned}$$

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 Sec. 6.3.

Sums Sum types can be viewed as a generalization of the boolean case. They are reduced to the canonical form ϕ ? `inl` e_l : `inr` e_r using:

$$\begin{aligned} & \phi \,?\, (\phi_1 \,?\, \texttt{inl}\ e_{l1} \;:\; \texttt{inr}\ e_{r1}) \;:\; (\phi_2 \,?\, \texttt{inl}\ e_{l2} \;:\; \texttt{inr}\ e_{r2}) \\ & \rightarrow \texttt{ite}\ \phi\ \phi_1\ \phi_2 \,?\, (\texttt{inl}\ \phi \,?\, e_{l1} \;:\; e_{l2}) \;:\; (\texttt{inr}\ \phi \,?\, e_{r1} \;:\; e_{r2}) \end{aligned}$$

This allows us to replace the *st-case-phi* rule with one that does not duplicate the work of the case expression:

$$\texttt{case}\ (\phi \,?\, \texttt{inl}\ e_1 \;:\; \texttt{inr}\ e_2)\ f_1\ f_2 \rightarrow \phi \,?\, (f_1\ e_1) \;:\; (f_2\ e_2)$$

Functions The normal form for functions are functions:

$$\phi \,?\, f_1 \;:\; f_2 \rightarrow \lambda x \,.\, \phi \,?\, (f_1\ x) \;:\; (f_2\ x)$$

This eliminates the need for *st-beta-phi* and creates additional opportunities for sharing when the same function is applied multiple times.

6.2 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 non-termination (Sec. 5.1) introduces arbitrary values for black box subterms. Using explicitly unreachable expressions instead has the effect of selecting the value that simplifies the approximation the most.

6.3 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:

$$(\phi_1 \,?\, 1 \;:\; 2) < (\phi_2 \,?\, 2 \;:\; 3)$$

Without support for integers, the `<` operator must be pushed into the branches of the ϕ-conditional for evaluation:

$$\begin{aligned} & \phi_1 \,?\, (\phi_2 \,?\, (1 < 2) \;:\; (1 < 3)) \\ & \quad : \; (\phi_2 \,?\, (2 < 2) \;:\; (2 < 3)) \end{aligned}$$

This exponential duplication of work can be avoided by representing the integers and operations on them in an SMT formula. In this case, we translate `<` to the SMT solver's `lt`:

$$\texttt{lt}\ (\texttt{ite}\ \phi_1\ 1\ 2)\ (\texttt{ite}\ \phi_2\ 2\ 3)$$

To fully exploit an underlying theory we must have direct access to free variables of the corresponding types. Our search primitives encapsulate this for boolean variables:

```
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 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.

6.4 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:

$$\phi_x \wedge \texttt{false} \rightarrow \texttt{false}$$
$$\texttt{ite}\ \texttt{false}\ \phi_x\ \phi_y \rightarrow \phi_y$$
$$\texttt{ite}\ \phi_x\ \phi_y\ \phi_y \rightarrow \phi_y$$

6.5 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.

7. Smten Compiler and Runtime

An efficient implementation of the functional programing aspects of the Smten language requires significant development effort that we did not wish to replicate. We cannot implement Smten as an embedded language in Haskell and compile using a modern compiler, such as the Glasgow Haskell Compiler (GHC) [15], however, because our implementation approach introduces ϕ-conditional expressions in the core syntax in fundamental way. Instead, we make use of GHC in two ways. First, we leverage the GHC runtime by implementing the Smten intermediate representation directly in Haskell. This is not as simple as a trivial syntax-directed translation because each data type must be translated into a new data type, and all `if` and `case` expressions must be rewritten to support the ϕ-conditional expression.

Second, to leverage GHC's package management system, parser, type inference, type checking, and module support, we implement the Smten compiler as a GHC plugin. The Smten plugin takes its input in the form of GHC's core intermediate representation. As a result our Smten implementation supports the entirety of Haskell98 and the majority of Haskell programs accepted by GHC. Some extensions are not fully supported yet, most notably the Haskell foreign function interface, because it requires a more complicated translation.

The Smten library borrows heavily from the Haskell Prelude; it supports most of the standard library functions and libraries, including arrays, maps, and monad transformers. We keep the Smten Prelude distinct from the Haskell Prelude because the GHC implementation uses extensions for performance and optimizations that are not fully supported in Smten. We hope to unify the two in the future after improving our translation from the core intermediate representation and optimization passes. Once unified, many existing user-contributed libraries for Haskell should work out-of-the-box in Smten.

8. Evaluation

To evaluate Smten, we used it to implement three complex satisfiability-based search applications from different domains. This includes a reimplementation of the HAMPI string constraint solver [20], a model checker based on k-induction, and a reimplementation of the sketch tool for program synthesis [33].

In this section we compare our implementations against state-of-the-art tools and discuss how it was relatively easy to construct our own implementations that achieve performance comparable to the originals using many fewer lines of source code thanks to Smten. The source code for our implementation of Smten and each of these applications can be found on github at `http://github.com/ruhler/` in the `smten` and `smten-apps` repositories respectively.

8.1 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 fixed-size context-free languages.

A HAMPI input consists of the definitions for regular expressions and context free grammars (CFGs), bounded-size string variables, and predicates on these strings referencing the regular expressions and grammars. 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 implementation of HAMPI is implemented in about 20K lines of Java and uses the STP [14] SMT solver. In contrast, our implementation of SHAMPI is around 1K lines of Smten code, including the lexer and parser for the string constraint language. This is an order of magnitude reduction in code size.

We capture HAMPI's algorithmic sharing using a straight forward memoization technique. The numbers presented here are an improvement of previously published work [37]

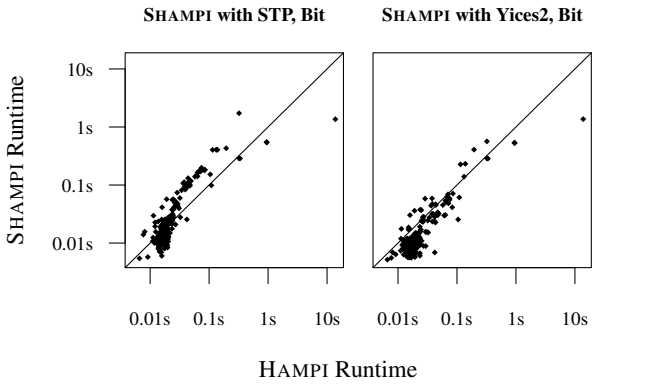

Figure 10. HAMPI compared to SHAMPI.

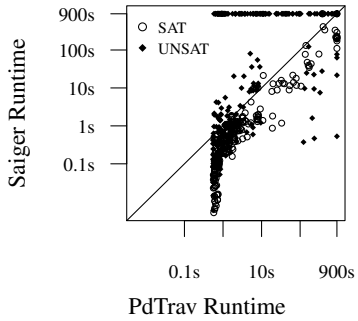

Figure 11. Saiger compared to PdTrav.

in that the whole of SHAMPI is now implemented in the Smten language.

Our initial development of SHAMPI required approximately three weeks of effort, including efforts required to understand the tool and important optimizations required to run effectively. 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.

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 [41] SMT solver, which performs notably better. Figure 10 compares 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. We ran revision 46 of a single HAMPI server instance for all runs of all tests on HAMPI to amortize the JVM startup cost; this also allows code specialization to happen in later instances of each input biasing the results towards HAMPI slightly.

Assuming our implementation includes the same algorithms and optimizations for generating the queries, as we believe to be the case, the left graph in Fig. 10 represents the overheads associated with using the more general Smten framework for solving the string constraint problem. The right graph represents the benefits the framework provides in requiring a trivial amount of effort to experiment with different backend solvers.

8.2 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 and property checking required by the 2010 hardware model checking competition [18]. It 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 [31]. 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 Aiger format to the core algorithm, and approximately 70 lines for parsing options and calling the core algorithm. In total, Saiger is less than 400 lines of Smten code, and took on the order of one week of effort to implement.

Figure 11 shows the performance of Saiger on the benchmarks from the 2010 hardware model checking competition compared to PdTrav [4]. PdTrav is the best model checker in the 2010 competition able to identify both SAT and UNSAT models that we were able to run ourselves. 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 to the property were found, labeled as SAT benchmarks. There are a number of UNSAT benchmarks on which our model checker exceeds the 900 second timeout limit which 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 exactly the descriptions from the paper presenting the algorithm. This means 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

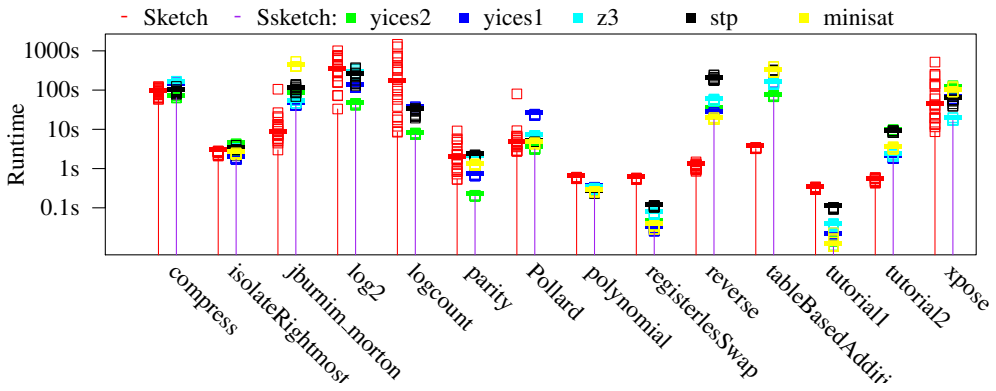

Figure 12. Sketch compared to Ssketch on gallery benchmarks.

could be reused unmodified for many different models, and perhaps even for software model checking.

8.3 Sketch Program Synthesis

To showcase Smten's ability to express complex multi-query computations we implemented the core subset of the Sketch [33] 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, the following is a sketch of an optimized function for isolating the rightmost unset bit of a word, along with an unoptimized specification:

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

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;
}
```

Here, the 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] i0sketch (bit[W] x) implements isolate0 {
  return ~(x + 0) & (x + 1);
}
```

The core of Sketch is realized by a Counter-Example Guided Inductive Synthesis (CEGIS) procedure [32]. 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 [6], optimize database applications [7], and factoring of functionalities [40].

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. The majority of effort required for us to implement Ssketch is devoted to implementing the semantics and features of the Sketch language, and not effort specifically devoted to the program synthesis or query generation aspects of Ssketch. The effort is comparable to the effort that would be required to develop a 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 12 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 Fig. 12, 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 by the backend solver used.

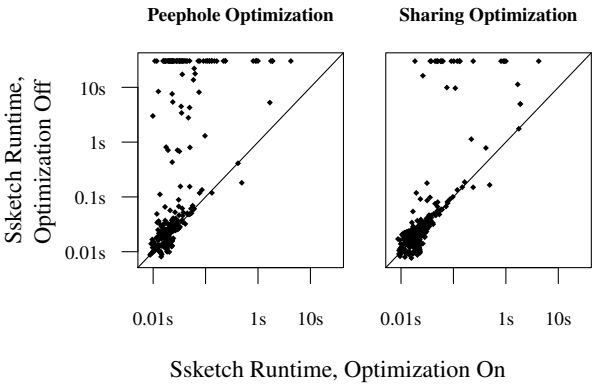

Figure 13. Impact of Smten optimizations on Ssketch.

The takeaway from Fig. 12 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 Ssketch, this was the case for all of the benchmarks shown in Fig. 12, and is typical of development of an SMT-based tool. It is fairly straight-forward, 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 based on the original array rather than after truncating an array by a potentially 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.

8.4 Optimizations

It is difficult to quantify the effects of the individual Smten optimizations discussed in Sec. 6. 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 13 compares the performance of Ssketch with and without peephole optimizations (which includes pruning based on unreachable expressions) and sharing optimizations in the Smten implementation on 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* effected by the optimizations, be it the peephole optimizations or preservation of sharing.

9. Related Work

A variety of approaches have been explored for making it easier to develop high-performance solutions to search problems in general. We primarily focus here on those relating to use of SAT and SMT solvers and those relating to use of functional language like Haskell for describing search problems.

The background theories of SMT solvers allow the developer to directly express their queries at a much higher level than SAT solvers, which SMT solvers take advantage of to improve performance. Some SMT solvers such as Z3 [11] and Yices [12] support record types and lambda terms. 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.

Libraries such as the python and F# bindings to the Z3 solver simplify interfacing with SMT solvers from general purpose languages, but it is still up to developers to implement optimizations on top of these libraries when constructing queries. More sophisticated interfaces to SMT have been built using a Domain Specific Embedded Language (DSEL) [25] approach. These DSELs [1, 13, 22, 34] provide a full metaprogramming layer to generate SMT queries. While it is possible to embed a domain specific language in a language such as Haskell that takes advantage of non-strict evaluation in the construction of queries, this approach is limited to optimizing the primitive types and operations at the lowest level, and fails to address the much broader scope of possible optimizations and structure in the metaprogramming layer, and in particular for user defined data types.

SAT/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 [35] and HALO [38] generate and run queries at compile time and are used for static analysis instead of the applications Smten is targeted at, where queries are generated and evaluated at runtime and optimizations based on dynamic data are much more important. Kuncak et al. [24] statically generate queries for SCALA that can depend on runtime parameters, but the structure of the query is fixed statically.

Our abstraction-refinement procedure for handling potentially non-terminating search spaces is similar to that used in Leon [35], which uses incremental unrolling of functions to

avoid non-termination, and encodes the reachability of approximated terms in the SMT formula itself. Our approach is distinct in that it does not rely on support for functions in the underlying solver, and we do not introduce any new free variables in the query to represent the approximated terms in the formula.

Kaplan [23] and Rosette [36] both are intended for development of applications that execute SAT and SMT queries dynamically. 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. Unlike Rosette, Smten does not require programs to be self-finitizing.

Our use of a set monad for describing search spaces is consistent with work by Hughes and O'Donnell [17] showing that sets are a good way to express non-deterministic computation in a functional programming language while preserving referential transparency. We extend that to include support for backtracking search and make use of monadic computation [39] for describing the sets.

Other approaches have been suggested to provide better functionality and performance for monadic search than the list monad. Work by Kiselyov et al. [21] and monadic constraint programming [30] provide more flexibility and control over how the search is performed. Neither of these approaches is targeted towards using SAT and SMT to improve the performance of the search.

Our work on Smten is closely related to Functional Logic Programming languages like Curry [16]. We believe describing search spaces in Smten will be more familiar to programmers than Curry, because the search spaces are described with functions that operate as they would on concrete data. This is in contrast to Curry, which changes the semantic interpretation of clauses in a function. It is likely techniques from Curry could be adapted to further improve the performance of Smten's runtime.

The idea of describing search computation in terms of set operations while using a different internal representation to improve performance is similar to database queries. Database queries are expressed in terms of direct operations on database tables, but are instead executed using sophisticated data structures internal to databases. Work on database query optimization [19, 26, 29] may be applicable to Smten, though it is not clear how easy it would be to model the cost of a SAT or SMT query to identify the best query to generate.

10. Discussion

One challenge we discovered while using Smten to develop complex applications is that programs are much more sensitive to small algorithmic changes than normal Haskell programs where all expressions are fully concrete. In the construction of queries, Smten conceptually evaluates a function by executing the union of all paths of the function. This means the number of possible paths and the complexity of each plays an important role in determining performance, not just the average path taken as in standard computation.

The best representation of a function for search in Smten may be different than the best representation for a concrete computation. Attempts to allow fast paths so common cases may immediately return can actually hurt performance when all paths must be considered, because they increase the number of paths that must be considered.

For example, consider the following list update functions that exemplify a number of cases we encountered when programming with Smten:

```
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
```

Here `update1` implements a short circuit; when the position to update is reached, the tail 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 paths. Each element 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 paths, because there is a separate computation path for each position n could take, and each of those n paths has on the order of n elements to check for the update.

We believe it is not too onerous to expect 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 paths must be evaluated to construct a query. This is also a reason why a non-strict language like Haskell works well for our approach, because functions written to take advantage of lazy evaluation can be efficient for both the average case in concrete computation, and the union of all cases for computations involving partially concrete expressions.

10.1 Future Work

Smten currently works very well when search spaces are constructed with explicit structure, but not as well at inferring additional structure of a search space from constraints placed on it. For example, recall the `aStr` search space presented in Sec. 3.3 that represents the set of strings of all lengths consisting of just the character `'a'`:

```
aStr :: Space String
aStr = union (single []) (map (λs → 'a':s) aStr)
```

An alternative approach to describing this set is to restrict the set of all strings to those that contain only the character 'a':

```
str :: Space String
str = do
  c ← choose ['a'..'z']
  union (single []) (map (λs → c:s) str)

aStr :: Space string
aStr = do
  s ← str
  if (all (== 'a') s)
    then single s
    else empty
```

Currently our implementation of Smten fails to infer from the second approach that every element of the partially concrete string must be 'a'. We would like to implement better implied value concretization [5], to make both approaches perform equally well. This is complicated to support efficiently in the presence of our existing optimizations and while preserving an appropriate level of sharing between expressions that appear in multiple different contexts.

11. Conclusion

We have presented Smten, a non-strict, 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 to achieve results comparable to standard SMT-based tools. We hope the consequences of Smten will be many more high-performance satisfiability-based search applications, especially in specialized domains that previously could not justify investing the high costs required to develop an efficient SMT-based tool, and joint effort in producing high quality library codes reused across a wide range of different satisfiability-based search applications.

Acknowledgments

This work 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 authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] S. Agarwal. Functional SMT solving: A new interface for programmers. Master's thesis, Indian Institute of Technology Kanpur, June 2012.

[2] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, pages 825–886. IOS Press, 2009.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W. R. 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.

[4] 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.

[5] C. Cadar, D. Dunbar, and D. 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.

[6] A. Cheung, A. Solar-Lezama, and S. Madden. Using program synthesis for social recommendations. *CoRR*, abs/1208.2925, 2012.

[7] A. Cheung, A. Solar-Lezama, and S. 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.

[8] S. 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.

[9] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960. ISSN 0004-5411.

[10] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. ISSN 0001-0782.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. 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.

[12] B. Dutertre and L. D. Moura. The Yices SMT solver. 2006.

[13] L. Erkok. http://hackage.haskell.org/package/sbv-3.0, 2014.

[14] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer Berlin / Heidelberg, 2007.

[15] T. Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[16] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

[17] J. Hughes and J. ODonnell. Expressing and reasoning about non-deterministic functional programs. In K. Davis and J. Hughes, editors, *Functional Programming*, Workshops in Computing, pages 308–328. Springer London, 1990.

[18] HWMCC10. 2010 hardware model checking competition. `http://fmv.jku.at/hwmcc10/` , 2010.

[19] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, June 1984. ISSN 0360-0300.

[20] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. 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.

[21] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. 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.

[22] A. Köksal, V. Kuncak, and P. Suter. Scala to the power of Z3: Integrating SMT and programming. In N. Bjrner and V. Sofronie-Stokkermans, editors, *Automated Deduction CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 400–406. Springer Berlin / Heidelberg, 2011.

[23] A. S. Köksal, V. Kuncak, and P. 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.

[24] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. *SIGPLAN Not.*, 45(6):316–329, June 2010. ISSN 0362-1340.

[25] D. Leijen and E. 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.

[26] M. Liu, Z. G. Ives, and B. T. Loo. Query optimization as a datalog program.

[27] S. Marlow and Others. Haskell 2010 language report. http://www.haskell.org/onlinereport/haskell2010, Apr. 2010.

[28] K. McMillan. Interpolation and SAT-based model checking. In J. Hunt, WarrenA. and F. Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2003.

[29] D. G. Murray, M. Isard, and Y. Yu. Steno: Automatic optimization of declarative queries. *SIGPLAN Not.*, 46(6):121–131, June 2011. ISSN 0362-1340.

[30] T. Schrijvers, P. Stuckey, and P. Wadler. Monadic constraint programming. *J. Funct. Program.*, 19(6):663–697, Nov. 2009. ISSN 0956-7968.

[31] M. Sheeran, S. Singh, and G. 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.

[32] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008. AAI3353225.

[33] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. 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.

[34] D. Stewart. http://hackage.haskell.org/package/yices-painless-0.1.2, Jan. 2011.

[35] P. Suter, A. S. Köksal, and V. 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.

[36] E. Torlak and R. 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.

[37] R. Uhler and N. Dave. Smten: Automatic translation of high-level symbolic computations into SMT queries. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 678–683. Springer Berlin Heidelberg, 2013.

[38] D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. 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.

[39] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 12 1992.

[40] K. Yessenov, Z. Xu, and A. Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In *OOPSLA*, pages 65–82, 2011.

[41] Yices2. http://yices.csl.sri.com/index.shtml, Aug. 2012.