Smten with Satisfiability-Based Search

THREADS 2014

Nirav Dave
SRI International

Richard Uhler
MIT-CSAIL-CSG

November 13, 2014

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 Department of Defense, or the National Science Foundation.

Satisfiability-Based Combinatorial Search

Combinatorial Search Problems

- Architectural Extraction
- Automatic Test Generation
- Automatic Theorem Proving
- Logic Synthesis
- Model Checking

- Program Synthesis
- Quantum Logic Synthesis
- String Constraint Solving
- Software Verification
- Sudoku

Satisfiability-Based Combinatorial Search

Combinatorial Search Problems

Solved using **Satisfiability** (**SAT**) and **Satisfiability Modulo Theories** (**SMT**) solvers.

- Architectural Extraction [Dave et al., 2011]
- Automatic Test Generation KLEE [Cadar et al., 2008]
- Automatic Theorem Proving Dafny [Leino, 2010]
- Logic Synthesis [Mishchenko et al., 2011]
- Model Checking [Biere et al., 1999, McMillan, 2003, Sheeran et al., 2000]
- Program Synthesis Sketch [Solar-Lezama et al., 2006]
- Quantum Logic Synthesis [Hung et al., 2004]
- String Constraint Solving HAMPI [Kiezun et al., 2009]
- Software Verification HALO [Vytiniotis et al., 2013]
- Sudoku [Weber, 2005]

Satisfiability-Based Combinatorial Search

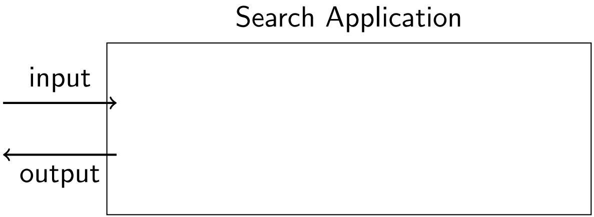
Combinatorial Search Problems

Solved using **Satisfiability (SAT)** and **Satisfiability Modulo Theories (SMT)** solvers.

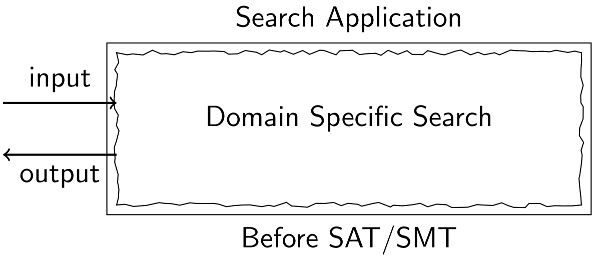
- Architectural Extraction [Dave et al., 2011]
- Automatic Test Generation KLEE [Cadar et al., 2008]
- Automatic Theorem Proving Dafny [Leino, 2010]
- Logic Synthesis [Mishchenko et al., 2011]
- Model Checking [Biere et al., 1999, McMillan, 2003, Sheeran et al., 2000]
- Program Synthesis Sketch [Solar-Lezama et al., 2006]
- Quantum Logic Synthesis [Hung et al., 2004]
- String Constraint Solving HAMPI [Kiezun et al., 2009]
- Software Verification HALO [Vytiniotis et al., 2013]
- Sudoku [Weber, 2005]

Many of these are key applications in formally verifying functionality/security properties

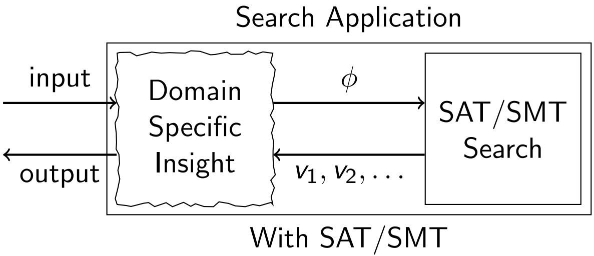
Why Use SAT and SMT?



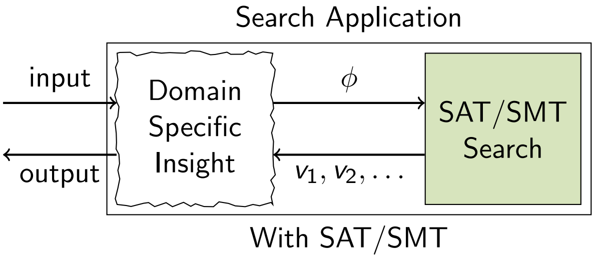
Why Use SAT and SMT?



Why Use SAT and SMT?



Why Use SAT and SMT?

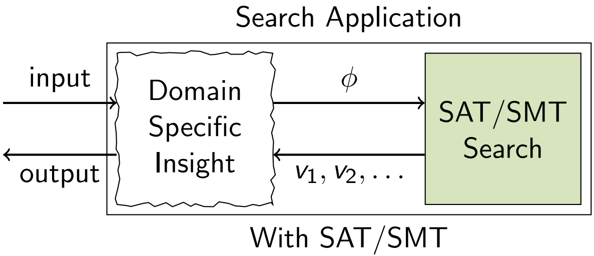


Reuse shared, expert search procedure

SAT $\quad \exists v_1, v_2, \ldots, v_n.\ \phi(v_1, v_2, \ldots, v_n) = \texttt{true}$?

$$\phi \quad ::= \quad \texttt{true} \mid \texttt{false} \mid v \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

Why Use SAT and SMT?

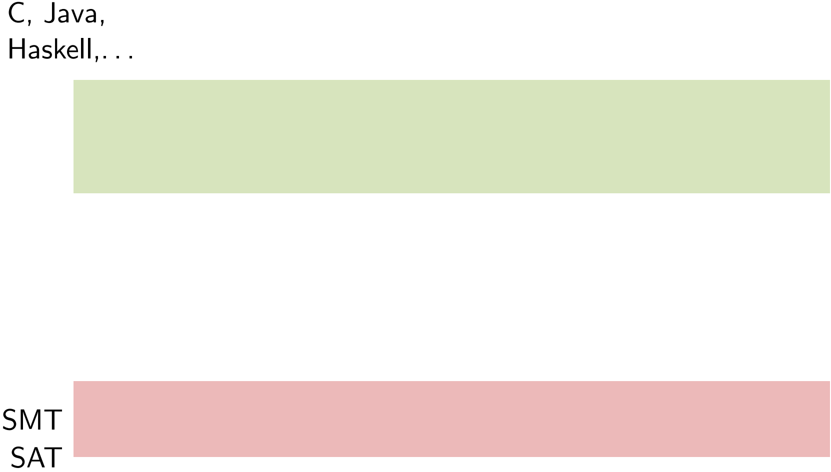


Reuse shared, expert search procedure

SAT/SMT $\exists v_1, v_2, \ldots, v_n.\ \phi(v_1, v_2, \ldots, v_n) = \texttt{true}$?

$$
\begin{array}{lcl}
\phi & ::= & \texttt{true} \mid \texttt{false} \mid v \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\
\text{Core} & \mid & \phi_1 = \phi_2 \mid \texttt{ite}\ \phi_1\ \phi_2\ \phi_3 \\
\text{Integer} & \mid & n \mid \phi_1 + \phi_2 \mid \phi_1 - \phi_2 \mid \phi_1 < \phi_2 \mid \cdots \\
\text{Bit Vector} & \mid & \texttt{bvlit}\ n \mid \texttt{bvadd}\ \phi_1\ \phi_2 \mid \texttt{bvsub}\ \phi_1\ \phi_2 \mid \cdots \\
\text{Array} & \mid & \texttt{read}\ \phi \mid \texttt{write}\ \phi_1\ \phi_2 \\
\underline{\text{theory}} & \cdots &
\end{array}
$$

SAT-Based Approach to Search



SAT-Based Approach to Search

C, Java,
Haskell,...

`ab??cd matches (ab)*(cd)*`

SMT
SAT

SAT-Based Approach to Search

C, Java,
Haskell,...

`ab??cd matches (ab)*(cd)*`

```
"abaacd"
"ababcd"
   ...
"abddcd"
```

SMT
SAT

SAT-Based Approach to Search

C, Java,
Haskell,...

`ab??cd matches (ab)*(cd)*`

"abaacd"		False
"ababcd"	RegEx.match ⟶	True
...		...
"abddcd"		False

SMT
SAT

SAT-Based Approach to Search

C, Java,
Haskell,...

`ab??cd matches (ab)*(cd)*`

"abaacd"		False	
"ababcd"	$\xrightarrow{\texttt{RegEx.match}}$	True	"ababcd"
...		...	
"abddcd"		False	

SMT
SAT

SAT-Based Approach to Search

C, Java,
Haskell,...

`ab??cd matches (ab)*(cd)*`

"abaacd"		False	
"ababcd"	RegEx.match ⟶	True	"ababcd"
...		...	
"abddcd"		False	

SMT
SAT

SAT-Based Approach to Search

C, Java,
Haskell,...

`ab??cd matches (ab)*(cd)*`

"abaacd"		False	
"ababcd"	RegEx.match $\longrightarrow$	True	"ababcd"
...		...	
"abddcd"		False	

$\downarrow$ encode

SMT
SAT

$97, 98, x_1,$
$x_2, 99, 100$

SAT-Based Approach to Search

C, Java,
Haskell,...

`ab??cd matches (ab)*(cd)*`

`"abaacd"`		`False`	
`"ababcd"`	`RegEx.match` $\longrightarrow$	`True`	`"ababcd"`
...		...	
`"abddcd"`		`False`	

encode

`mymatch` $\left\{ \begin{array}{c} \text{Regex.match} \\ \text{unrolled and encoded} \end{array} \right.$

SMT
SAT

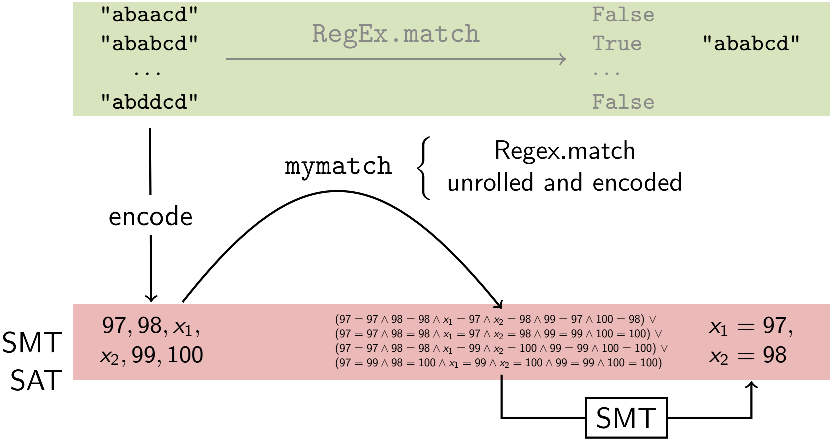
$97, 98, x_1, x_2, 99, 100$

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

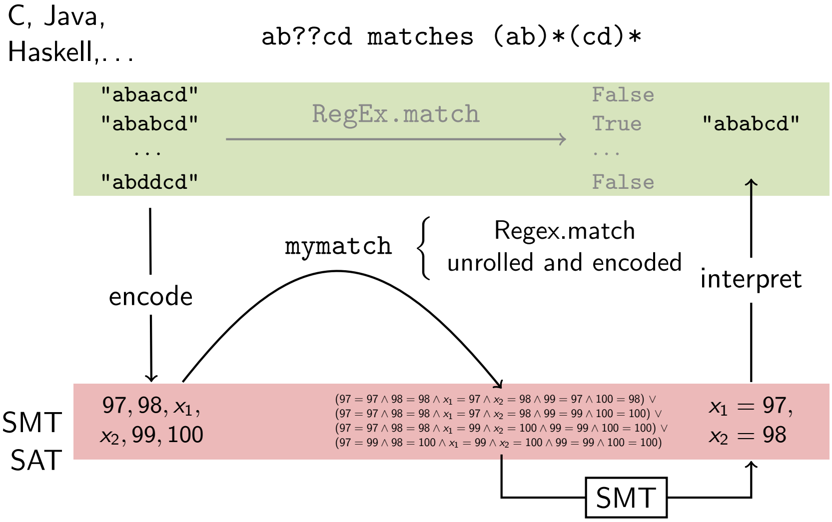
SAT-Based Approach to Search

C, Java, Haskell, ...

`ab??cd matches (ab)*(cd)*`



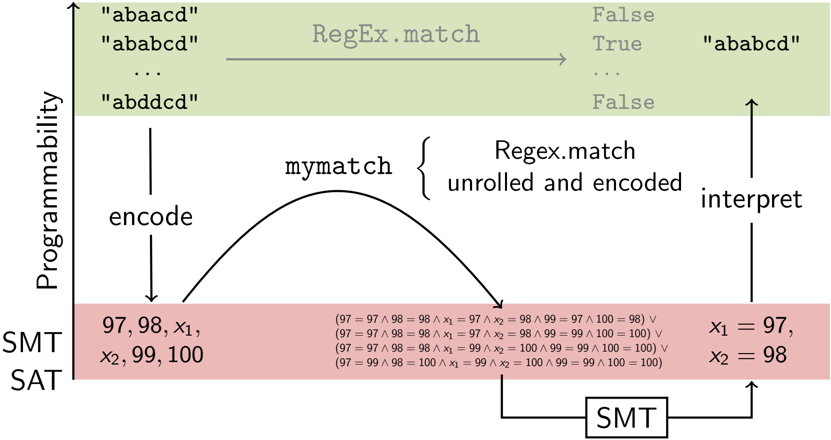
SAT-Based Approach to Search



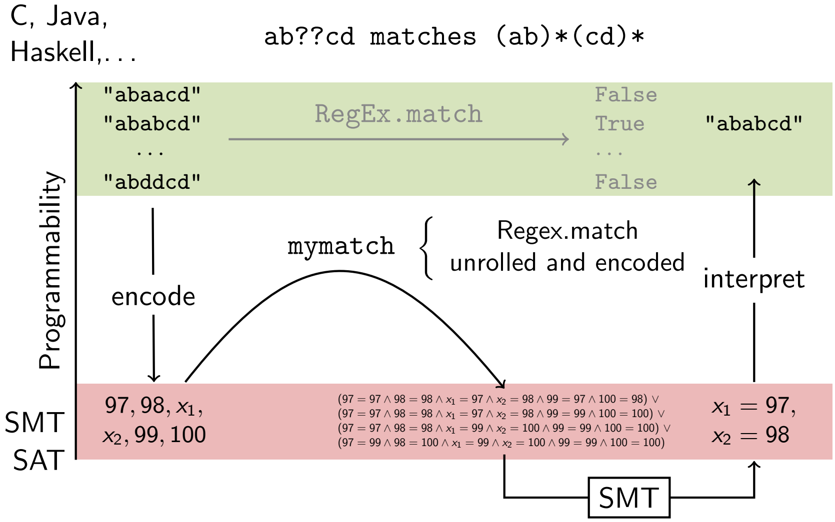
The Trouble with Using SAT/SMT

C, Java,
Haskell,. . .

`ab??cd matches (ab)*(cd)*`

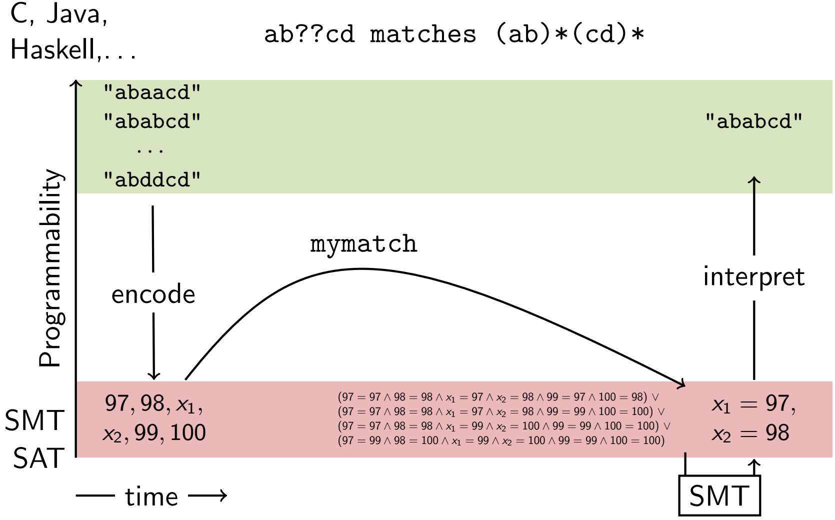


The Trouble with Using SAT/SMT

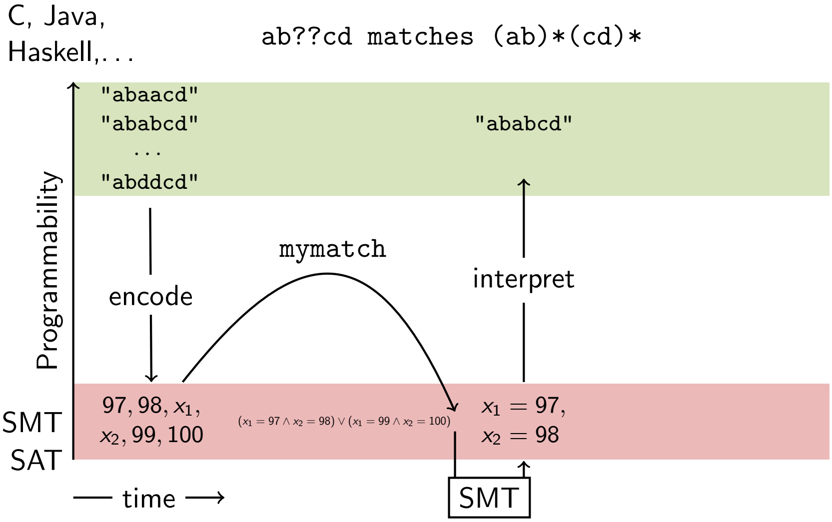


Indirection Code is harder to develop, read, understand, debug, and maintain.

The Trouble with Using SAT/SMT

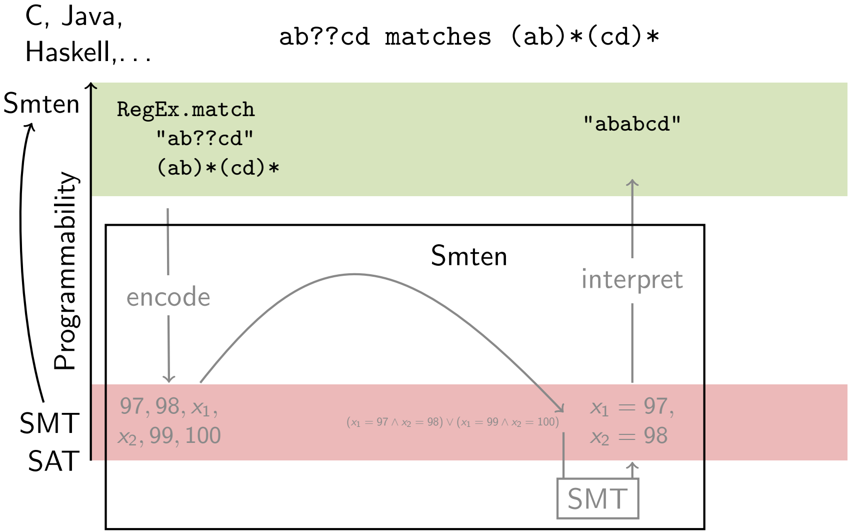


The Trouble with Using SAT/SMT



Indirection Developer's code must optimize queries as they are constructed.

Smten with SAT-Based Search



Smten with SAT-Based Search

C, Java,
Haskell,. . .

Smten

Programmability

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

Smten

Benefits of Raising the Level of Abstraction

Development

- Directly express high-level operations
- Easier to maintain modularity and enable reuse
- Less code, faster debugging (10-100x)

Performance

- Automatically apply simple but key low-level optimizations
- Dramatically lowers cost of radical algorithmic changes
- Allows easy migration to different SMT solvers

The Smten Language

High-level Goal

Express a query using general purpose programming features of the form:

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

Our Solution: Use...

Haskell Functional programming language for general purpose programming features

$+$ Sets To describe the search space.

- Sets provide a clean way to describe choice while preserving referential transparency [Hughes and ODonnell, 1990].

Smten Search Spaces

```
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
```

Smten Search Spaces

```
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
```

Example A: {True, False}

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

Smten Search Spaces

```
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
```

Example B: {"foo", "sludge"}

```
strB :: Space String
strB = union (single "foo") (single "sludge")
```

Smten Search Spaces

```
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
```

Example C:

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

```
strC :: Space String
strC = do
  s1 ← union (single "hi") (single "bye")
  s2 ← strB
  single (s1 ++ "␣" ++ s2)
```

Smten Search Spaces

```
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
```

Example D:

{"hi foo", "hi sludge", ~~"bye foo"~~, "bye sludge"}

```
strD :: Space String
strD = do
 s ← strC
 if length s /= 7
   then single s
   else empty
```

Smten Search Spaces

```
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
```

Example D:

{"hi foo", "hi sludge", ~~"bye foo"~~, "bye sludge"}

```
strD :: Space String
strD = do
 s ← strC
 if length s /= 7
   then single s
   else empty
```

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

Smten Search Spaces

```
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
```

Example E:

{"hi foo", "hi sludge", ~~"bye foo"~~, "bye sludge"}

```
strE :: Space String
strE = do
 s ← strC
 guard (length s /= 7)
 single s
```

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

Search Spaces for a String Constraint Solver

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

strs_regex :: RegEx → Space String
strs_regex r =
  case r of
    Empty → empty
    Epsilon → single ""
    Atom c → single [c]
    Concat a b → do
      sa ← strs_regex a
      sb ← strs_regex b
      single (sa ++ sb)
    Or a b → union (strs_regex a) (strs_regex b)
    Star x → union (single "") $ do
      sx ← strs_regex x
      sr ← strs_regex r
      single (sx ++ sr)

strs_regex_contains :: RegEx → [String] → Space String
strs_regex_contains r xs = do
  s ← strs_regex r
  guard (all (\x → contains x s) xs)
  single s
```

Searching a Search Space

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

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

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

- Solver is specified by name, independent of search space
- Searches for a single result
- Search is non-deterministic
- Searches cannot be nested
  - Nested search require quantifier alternation (for all x, there exists y)

A Smten String Constraint Solver

```
strsolve :: RegEx → [String] → IO ()
strsolve r xs = do
  result ← search yices2 (strs_regex_contains r xs)
  case result of
    Nothing → putStrLn "No␣Solution"
    Just x → putStrLn ("Solution: ␣" ++ show x)
```

Compilation to SMT Example

```
do s ← strs_regex /ab(c|d)(cd|ef)/
   if contains "de" s
      then single s
      else empty
```

Compilation to SMT Example

```
join $ map (\s → if contains "de" s
                 then single s
                 else empty)
           (strs_regex /ab(c|d)(cd|ef)/)
```

Compilation to SMT Example

```
join $ map (\s → if contains "de" s
                 then single s
                 else empty)
           (strs_regex /ab(c|d)(cd|ef)/)
```

Embed SMT variables in normal expressions:

Space Description: `strs_regex /ab(c|d)(cd|ef)/`

Set Interpretation

Smten Representation

$$\left\{\begin{array}{c}\texttt{"abccd"}\\ \texttt{"abcef"}\\ \texttt{"abdcd"}\\ \texttt{"abdef"}\end{array}\right\}$$

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

Compilation to SMT Example

```
join $ map (\s → if contains "de" s
                  then single s
                  else empty)
```

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

Embed SMT variables in normal expressions:

Space Description: `strs_regex /ab(c|d)(cd|ef)/`

Set Interpretation

Smten Representation

$$\left\{\begin{array}{c} \texttt{"abccd"} \\ \texttt{"abcef"} \\ \texttt{"abdcd"} \\ \texttt{"abdef"} \end{array}\right\}$$

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

Compilation to SMT Example

```
join $ map (\s → if contains "de" s
                   then single s
                   else empty)
```

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

Evaluate constraints on the compact representation:

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

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

Compilation to SMT Example

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

Evaluate constraints on the compact representation:

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

$$\texttt{map (contains "de")}\left\{\texttt{"ab}\begin{bmatrix} v_1 ?\texttt{c} \\ :\texttt{d}\end{bmatrix}\begin{bmatrix} v_2 ?\texttt{c} \\ :\texttt{e}\end{bmatrix}\begin{bmatrix} v_2 ?\texttt{d} \\ :\texttt{f}\end{bmatrix}\texttt{"}\right\}$$

$$= \left\{\left[\begin{array}{lll}(\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2) & ? & \texttt{True} \\ & : & \texttt{False}\end{array}\right]\right\}$$

Compilation to SMT Example

$$\left[\begin{array}{lll}(\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2) & ? & \texttt{single}\ \texttt{"ab}\begin{bmatrix} v_1?\texttt{c} \\ :\texttt{d}\end{bmatrix}\begin{bmatrix} v_2?\texttt{c} \\ :\texttt{e}\end{bmatrix}\begin{bmatrix} v_2?\texttt{d} \\ :\texttt{f}\end{bmatrix}\texttt{"} \\ & : & \texttt{empty}\end{array}\right]$$

Evaluate constraints on the compact representation:

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

$$\texttt{map (contains "de")}\ \left\{\texttt{"ab}\begin{bmatrix} v_1?\texttt{c} \\ :\texttt{d}\end{bmatrix}\begin{bmatrix} v_2?\texttt{c} \\ :\texttt{e}\end{bmatrix}\begin{bmatrix} v_2?\texttt{d} \\ :\texttt{f}\end{bmatrix}\texttt{"}\right\}$$

$$= \left\{\left[\begin{array}{lll}(\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2) & ? & \texttt{True} \\ & : & \texttt{False}\end{array}\right]\right\}$$

Compilation to SMT Example

$$\left[\begin{array}{lll}(\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2) & ? & \texttt{single}\ \texttt{"ab}\begin{bmatrix} v_1\texttt{?c} \\ \texttt{:d}\end{bmatrix}\begin{bmatrix} v_2\texttt{?c} \\ \texttt{:e}\end{bmatrix}\begin{bmatrix} v_2\texttt{?d} \\ \texttt{:f}\end{bmatrix}\texttt{"} \\ & : & \texttt{empty}\end{array}\right]$$

This gives us our SMT query:

$$\phi = (\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2)$$

And a solution, given a satisfying assignment assignment μ:

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

gives "abdef"

Compilation to SMT Example

$$\left[(\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2) \;\; \begin{array}{ll} ? & \texttt{single}\ \texttt{"ab} \begin{bmatrix} v_1 ?\texttt{c} \\ :\texttt{d} \end{bmatrix} \begin{bmatrix} v_2 ?\texttt{c} \\ :\texttt{e} \end{bmatrix} \begin{bmatrix} v_2 ?\texttt{d} \\ :\texttt{f} \end{bmatrix} \texttt{"} \\ : & \texttt{empty} \end{array} \right]$$

This gives us our SMT query:

$$\phi = (\neg v_1 \wedge \neg v_2) \vee (v_2 \wedge \neg v_2)$$

And a solution, given a satisfying assignment assignment μ:

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

gives "abdef"

See OOPSLA 2014 paper for details

Infinite Recursion in Search

Haskell is Turing-complete

- It is possible to expression infinite recursion in search

This is O.K.

- Useful for working with infinite search spaces and data types
  - e.g. Infinite length sequences of states, strings, programs
- Important for modularity
  - You should be able to use a potentially unbounded recursive function in search if it terminates on your inputs

What behavior should we expect?

To show $e \in s$: Construct only enough of s to see it contains e

To show $s = \emptyset$: Must construct the entire space s

Examples of Infinite Recursion in Search

```
ex1 = union (single ⊥) (single "foo")
```

```
    Just "foo", Just ⊥
```

Examples of Infinite Recursion in Search

```
ex1 = union (single ⊥) (single "foo")
```

Just "foo", Just ⊥

```
ex2 = union ⊥ (single "foo")
```

Just "foo"

Examples of Infinite Recursion in Search

```
ex1 = union (single ⊥) (single "foo")
```

Just "foo", Just ⊥

```
ex2 = union ⊥ (single "foo")
```

Just "foo"

```
ex3 = union ⊥ empty
```

Fails to terminate

Examples of Infinite Recursion in Search

```
ex1 = union (single ⊥) (single "foo")
```

Just "foo", Just ⊥

```
ex2 = union ⊥ (single "foo")
```

Just "foo"

```
ex3 = union ⊥ empty
```

Fails to terminate

```
strA = union (single "") (map ((:) 'a') strA)
```

Just "", Just "a", Just "aa", Just "aaa", ...

Examples of Infinite Recursion in Search

```
ex1 = union (single ⊥) (single "foo")
```

Just "foo", Just ⊥

```
ex2 = union ⊥ (single "foo")
```

Just "foo"

```
ex3 = union ⊥ empty
```

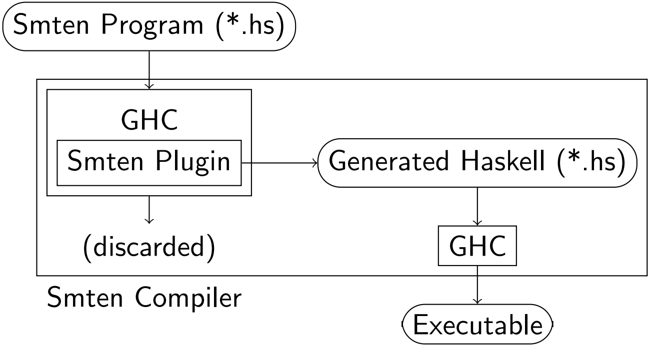
Fails to terminate

```
strA = union (single "") (map ((:) 'a') strA)
```

Just "", Just "a", Just "aa", Just "aaa", ...

We handle infinite recursion using a novel abstraction-refinement procedure for incrementally unrolling queries.

The Smten Compiler



- A modified version of the Glasgow Haskell Compiler (GHC)
- Supports full Haskell syntax and features
- Supports 5 backend solvers: Yices1, Yices2, STP, Z3, MiniSat
- Available open source: `http://github.com/ruhler/smten`

Evaluation

We've argued that Smten is more efficient, but let's see how it does in real applications

Shampi: String Constraint Solver

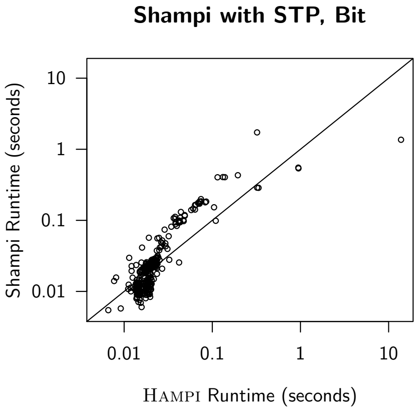
Reimplementation of Hampi [Kiezun et al., 2009]

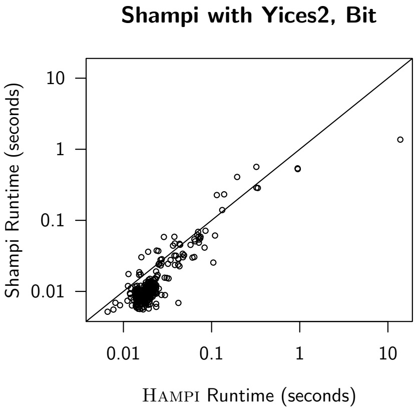
Development Experience

- Three weeks for initial implementation. Includes:
  - Understanding HAMPI input language
  - Optimizing Shampi
  - Working with early versions of Smten
- Revisited 6 months later:
  - Able to add new domain optimizations easily
  - Experimented with additional Char representations easily

Implementation	Lines of Source
HAMPI	20K Java
Shampi	1K Smten

Shampi Performance





- Run on all benchmarks from HAMPI paper
- The ability to easily experiment with different solvers overcomes the overheads of Smten

Saiger: Model Checker

AIGER Model Checker

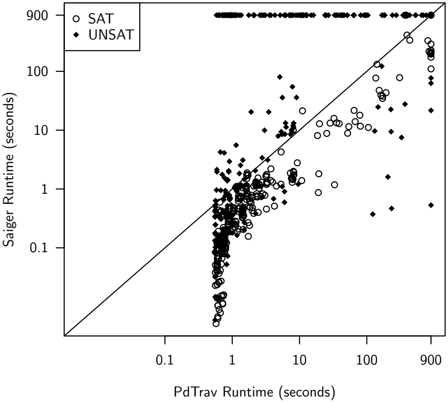
- K-induction based property checker [Sheeran et al., 2000]
- Accepts AIGER format (from the annual hardware model checking competition)

Development Experience

- One day to implement core algorithm
- One week to understand, implement, and integrate AIGER

Module	Lines	Description
Main	71	Command line parsing
PCheck	86	Core k-induction routine
Aiger	160	AIGER format parser and evaluator
AigerPCheck	50	Application of PCheck to AIGER format
Total	367	

Saiger Performance



- Compared against the latest version of `PdTrav`, a top place finisher of the 2010 hardware model checking competition
- Run on all 2010 competition benchmarks, with 900s timeout

SSketch: Program Synthesis

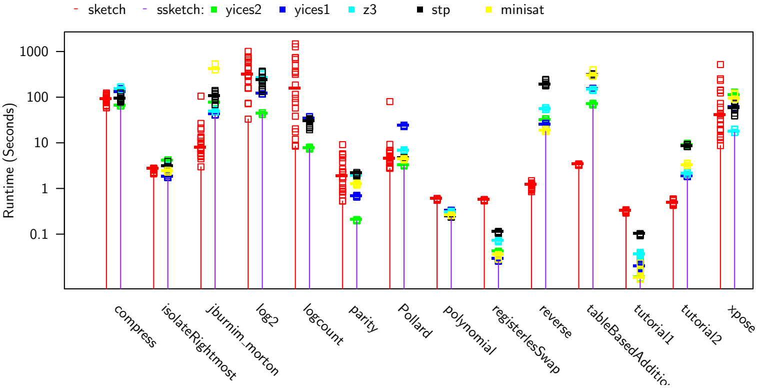
Reimplements Sketch [Solar-Lezama et al., 2006]

Development Experience

- 3-6 months of effort, considerably more than Shampi or Saiger
- Sketch has a rich language with many features
- Complexity of SSketch implementation due to complexity of the domain, not complexity of working with SAT/SMT or the search aspects of program synthesis.
- SSketch supports most Sketch features, but not:
    - stencils, uninterpreted functions, packages, chars, floats

Implementation	Lines of Source
Sketch	85K Java + 20K Cpp
SSketch	3K Smten

SSketch Performance



Conclusion

Smten

A language for SAT-based search which significantly reduces costs to develop practical SMT-based applications.

- Simplifies the SAT/SMT and makes it accessible to non-experts
- Automates tedious microoptimization efforts
- Facilitates design exploration
  - Choice of solver
  - Choice of background theory
  - Choice of encodings
  - Organization of search
- Encourages code reuse across applications

There's still plenty of interesting work to do on Smten and in developing applications.

Contact ⟨ndave@csl.sri.com⟩

GitHub Repo: `http://github.com/ruhler/smten`

Implementing Search

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

1. Evaluate s to canonical form: $\begin{bmatrix} \phi \ ? \ \texttt{single } e \\ \quad : \ \texttt{empty} \end{bmatrix}$

   (Details to follow)
2. Find a satisfying assignment μ to ϕ using the `slv` solver
3. If no μ found, $s = \emptyset$:

   $$\texttt{return Nothing}$$
4. Otherwise, $e[\mu] \in s$:

   $$\texttt{return (Just } e[\mu]\texttt{)}$$

Details of Syntax Directed Implementation

Smten Kernel Syntax

$$
\begin{array}{rcl}
e, f, s & ::= & x \mid \lambda x \,.\, e \mid f\ e \\
& \mid & \texttt{unit} \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
& \mid & \texttt{inl}_T\ e \mid \texttt{inr}\ e \mid \texttt{case}\ e\ f_1\ f_2 \\
& \mid & \texttt{fix}\ f \\
& \mid & \texttt{return}_{io}\ e \mid e \ \texttt{>>=}_{io}\ f \\
& \mid & \texttt{search}\ s \\
& \mid & \texttt{empty} \mid \texttt{single}\ e \mid \texttt{union}\ s_1\ s_2 \mid \texttt{map}\ f\ s \mid \texttt{join}\ s
\end{array}
$$

Details of Syntax Directed Implementation

Smten Kernel Syntax

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

Pure Evaluation with ϕ-Conditional

Primitive operations are pushed inside ϕ-conditional :

$$\begin{bmatrix}\phi \ ? \ f_1 \\ \ : \ f_2\end{bmatrix} \ e \quad \rightarrow \quad \begin{bmatrix}\phi \ ? \ (f_1 \ e) \\ \ : \ (f_2 \ e)\end{bmatrix}$$

$$\texttt{fst} \ \begin{bmatrix}\phi \ ? \ e_1 \\ \ : \ e_2\end{bmatrix} \quad \rightarrow \quad \begin{bmatrix}\phi \ ? \ (\texttt{fst} \ e_1) \\ \ : \ (\texttt{fst} \ e_2)\end{bmatrix}$$

$$\texttt{snd} \ \begin{bmatrix}\phi \ ? \ e_1 \\ \ : \ e_2\end{bmatrix} \quad \rightarrow \quad \begin{bmatrix}\phi \ ? \ (\texttt{snd} \ e_1) \\ \ : \ (\texttt{snd} \ e_2)\end{bmatrix}$$

$$\texttt{case} \ \begin{bmatrix}\phi \ ? \ e_1 \\ \ : \ e_2\end{bmatrix} \ f_1 \ f_2 \quad \rightarrow \quad \begin{bmatrix}\phi \ ? \ (\texttt{case} \ e_1 \ f_1 \ f_2) \\ \ : \ (\texttt{case} \ e_2 \ f_1 \ f_2)\end{bmatrix}$$

Remaining rules are <u>unmodified</u>. In particular:

$$(\lambda x \ . \ e) \ \begin{bmatrix}\phi \ ? \ e_1 \\ \ : \ e_2\end{bmatrix} \quad \rightarrow \quad e[\begin{bmatrix}\phi \ ? \ e_1 \\ \ : \ e_2\end{bmatrix} /x]$$

- With some optimizations, we can avoid pushing primitive operations inside ϕ-conditional expressions in most cases.

Evaluating s to Canonical Form

$$\texttt{empty} \rightarrow_s \begin{bmatrix} \texttt{false} \mathbin{?} \texttt{single}\ \bot \\ \quad : \texttt{empty} \end{bmatrix}$$

$$\texttt{single}\ e \rightarrow_s \begin{bmatrix} \texttt{true} \mathbin{?} \texttt{single}\ e \\ \quad : \texttt{empty} \end{bmatrix} \qquad (v \text{ fresh})$$

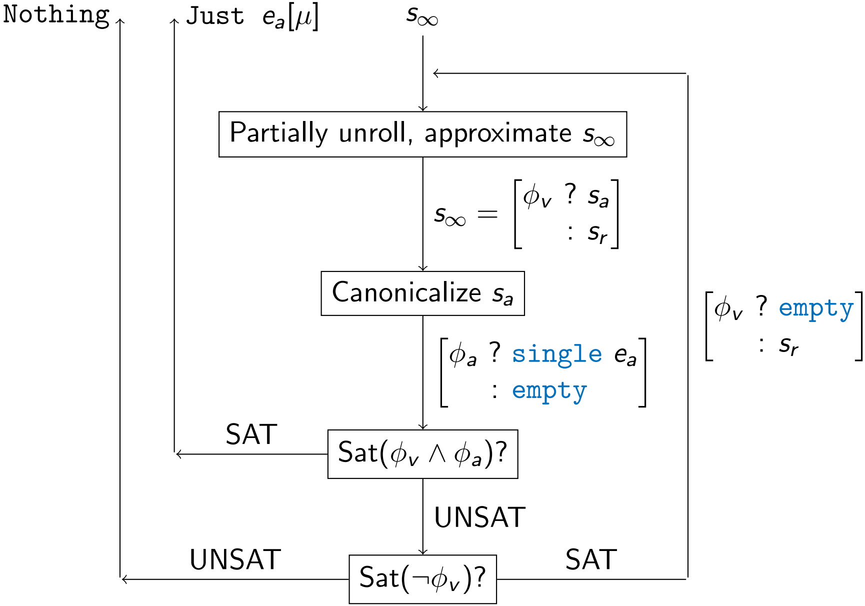
$$\texttt{union} \begin{bmatrix} \phi_1 \mathbin{?} \texttt{single}\ e_1 \\ \quad : \texttt{empty} \end{bmatrix} \begin{bmatrix} \phi_2 \mathbin{?} \texttt{single}\ e_2 \\ \quad : \texttt{empty} \end{bmatrix} \rightarrow_s \begin{bmatrix} \texttt{ite}\ v\ \phi_1\ \phi_2 \mathbin{?} \texttt{single} \begin{bmatrix} v \mathbin{?} e_1 \\ \quad : e_2 \end{bmatrix} \\ \quad : \texttt{empty} \end{bmatrix}$$

$$\texttt{map}\ f \begin{bmatrix} \phi \mathbin{?} \texttt{single}\ e \\ \quad : \texttt{empty} \end{bmatrix} \rightarrow_s \begin{bmatrix} \phi \mathbin{?} \texttt{single}\ (f\ e) \\ \quad : \texttt{empty} \end{bmatrix}$$

$$\texttt{join} \begin{bmatrix} \phi \mathbin{?} \texttt{single}\ s \\ \quad : \texttt{empty} \end{bmatrix} \rightarrow_s \begin{bmatrix} \phi \mathbin{?} s \\ \quad : \texttt{empty} \end{bmatrix}$$

$$\phi \mathbin{?} \begin{bmatrix} \phi_1 \mathbin{?} \texttt{single}\ e_1 \\ \quad : \texttt{empty} \end{bmatrix} : \begin{bmatrix} \phi_2 \mathbin{?} \texttt{single}\ e_2 \\ \quad : \texttt{empty} \end{bmatrix} \rightarrow_s \begin{bmatrix} \texttt{ite}\ \phi\ \phi_1\ \phi_2 \mathbin{?} \texttt{single} \begin{bmatrix} \phi \mathbin{?} e_1 \\ \quad : e_2 \end{bmatrix} \\ \quad : \texttt{empty} \end{bmatrix}$$

Abstraction-Refinement Procedure for Search



Shampi Code Breakdown

Shampi

Module	Lines	Description
CFG, SCFG, RegEx, Hampi	356	Data type definitions
Fix	103	Fix-sizing
Match	50	Core match algorithm (memoized)
Lexer, Grammar	303	Parser for HAMPI constraints
SChar	90	Char representations: Integer, Bit, Int, Char
Query	90	Query orchestration
Main	83	Command line parsing
Total	1059	lines of Smten

Original Hampi

Total	20,000	lines of Java

References

Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999).
Symbolic model checking without BDDs.
In Cleaveland, W. R., 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.

Cadar, C., Dunbar, D., and Engler, D. (2008).
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. USENIX Association.

Dave, N., Katelman, M., King, M., Arvind, and Meseguer, J. (2011).
Verification of microarchitectural refinements in rule-based systems.
In Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on, pages 61–71.

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

Hung, W. N. N., Song, X., Yang, G., Yang, J., and Perkowski, M. (2004).
Quantum logic synthesis by symbolic reachability analysis.
In Proceedings of the 41st Annual Design Automation Conference, DAC '04, pages 838–841, New York, NY, USA. ACM.

Kiezun, A., Ganesh, V., Guo, P. J., Hooimeijer, P., and Ernst, M. D. (2009).
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. ACM.

Leino, K. (2010).
Dafny: An automatic program verifier for functional correctness.
In Clarke, E. and Voronkov, A., editors, Logic for Programming, Artificial Intelligence, and Reasoning, volume 6355 of Lecture Notes in Computer Science, pages 348–370. Springer Berlin Heidelberg.

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

Mishchenko, A., Brayton, R., Jiang, J.-H. R., and Jang, S. (2011).
Scalable don't-care-based logic optimization and resynthesis.
ACM Trans. Reconfigurable Technol. Syst., 4(4):34:1–34:23.

Sheeran, M., Singh, S., and Stålmarck, G. (2000).
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. Springer-Verlag.

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. (2006).
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. ACM.

Vytiniotis, D., Peyton Jones, S., Claessen, K., and Rosén, D. (2013).
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. ACM.

Weber, T. (2005).
A SAT-based Sudoku solver.
In 12 th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2005, pages 11–15.