Progressive Reasoning over Recursively-Defined Strings

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar

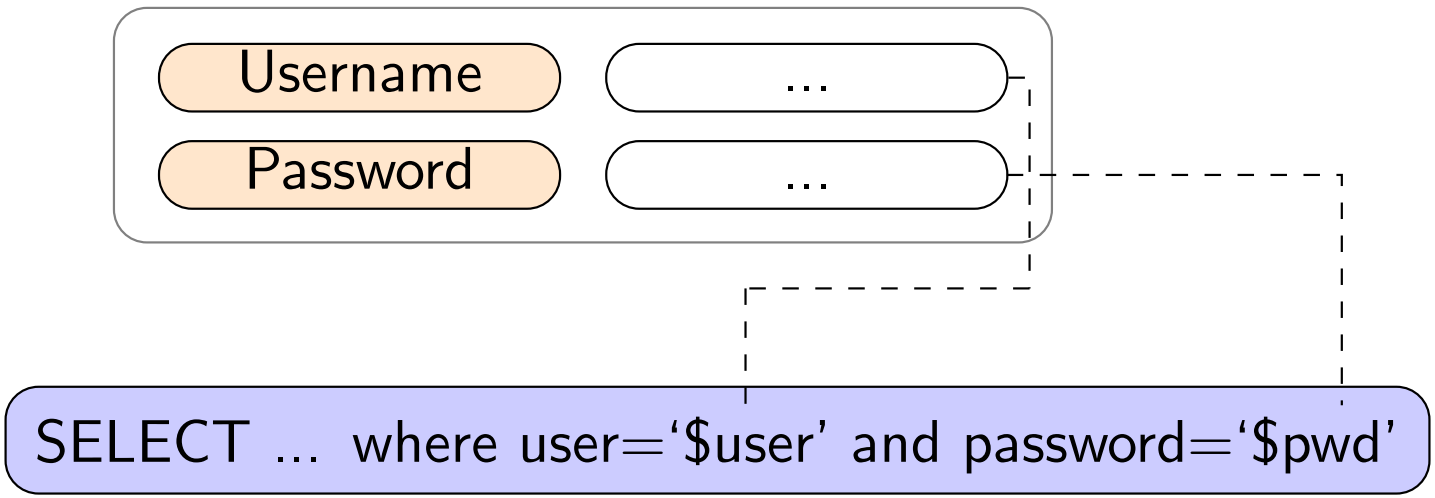
National University of Singapore (NUS)

July 20, 2016

Reasoning about String Data Type

- Motivated by the security of web applications, mobile applications, etc.

- E.g. web applications usually take string values as input, manipulate them, and then use them to construct database queries



Therefore,

- The most serious and critical vulnerabilities are involved in strings
- For example,
  - SQL injection: Username is `' OR 1=1--`
  - Cross-site scripting (XSS)
- To prevent insecure inputs from users
  - Security experts define attack patterns
    - e.g., every string that starts with `' OR 1=1--`
  - Developers use sanitizer functions to replace all substrings that match an attack pattern in the user input
    - e.g., via **replace** function in JavaScript, PHP

Recursively-Defined Strings

- For web applications, strings are unbounded
  - e.g., string is a primitive type in JavaScript, PHP, Python
  - unlike C, where a string is an array of characters
- **replace** function can be defined recursively:

$$Y = \textbf{replace}(X, r, Z) \stackrel{def}{=} (X \notin /.^\star r.^\star/ \wedge Y = X) \vee$$
$$(X = X_1 \cdot X_2 \cdot X_3 \cdot X_4 \wedge X_2 \cdot X_3 \in /r/ \wedge \textbf{length}(X_3) = 1 \wedge$$
$$X_1 \cdot X_2 \notin /.^\star r.^\star/ \wedge Y = X_1 \cdot Z \cdot Y_1 \wedge Y_1 = \textbf{replace}(X_4, r, Z))$$

  - general definition, where Z is a variable (e.g. as in PHP)
  - replacement of all substrings in X that match the pattern r
  - two cases
  - non-greedy version

The Challenge

- Solving *recursively-defined* string constraints is undecidable
  - E.g., string functions such as **replace** make the satisfiability problem undecidable (Büchi and Senger [1988]; Bjørner *et al.* [2009]).

State-of-The-Art Techniques

- Folding/Unfolding the recursive definitions
  - e.g., Z3-str2, CVC4, S3, Pisa, etc.
- However, they do not address the non-termination issue
  - Z3-str2 addresses non-termination in splitting overlapping string variables

Non-termination

Can happen when

- unfolding recursive functions such as **replace**
- splitting overlapping string variables
- dealing with Kleene star operator
  - $X \in r^\star$ is represented as $X = \mathbf{star}(r, N)$, where N is a fresh variable.
  - A recursive definition for **star** function:

$$X = \mathbf{star}(r, N) \stackrel{def}{=} (X = \text{“”}) \vee (X = r \cdot \mathbf{star}(r, M) \wedge N = M + 1)$$

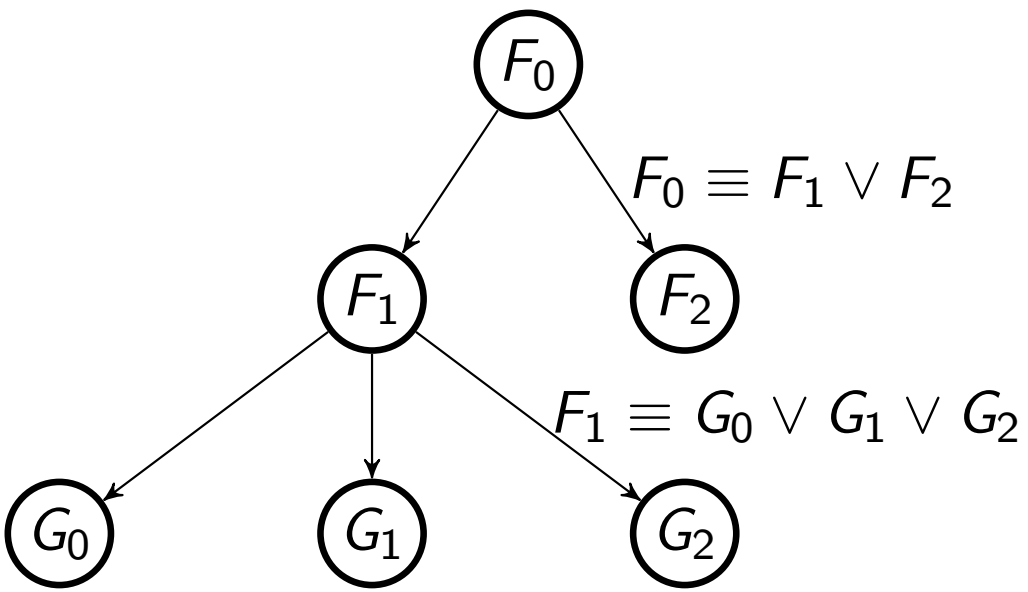
Example of Splitting

$$X \cdot \text{“}a\text{”} = \text{“}b\text{”} \cdot X$$

1. SPLIT: two disjuncts
   $X \cdot \text{“}a\text{”} = \text{“}b\text{”} \cdot X \wedge X = \text{“”}$ (I)
   $X \cdot \text{“}a\text{”} = \text{“}b\text{”} \cdot X \wedge X = \text{“}b\text{”} \cdot Y$ (II)
2. CONTRA: (I) leads to a contradiction $\text{“}a\text{”} = \text{“}b\text{”}$
3. SUBSTITUTE for (II):
   $\text{“}b\text{”} \cdot Y \cdot \text{“}a\text{”} = \text{“}b\text{”} \cdot \text{“}b\text{”} \cdot Y \wedge X = \text{“}b\text{”} \cdot Y$ (III)
4. SIMPLIFY for (III):
   $Y \cdot \text{“}a\text{”} = \text{“}b\text{”} \cdot Y \wedge X = \text{“}b\text{”} \cdot Y$ (IV)
5. SPLIT for (IV):
   $Y \cdot \text{“}a\text{”} = \text{“}b\text{”} \cdot Y \wedge X = \text{“}b\text{”} \cdot Y \wedge Y = \text{“”}$
   $Y \cdot \text{“}a\text{”} = \text{“}b\text{”} \cdot Y \wedge X = \text{“}b\text{”} \cdot Y \wedge Y = \text{“}b\text{”} \cdot Z$
6. CONTRA: ...
7. SUBSTITUTE: ...
8. SIMPLIFY:
   $Z \cdot \text{“}a\text{”} = \text{“}b\text{”} \cdot Z \wedge Y = \text{“}b\text{”} \cdot Z \wedge X = \text{“}b\text{”} \cdot Y$

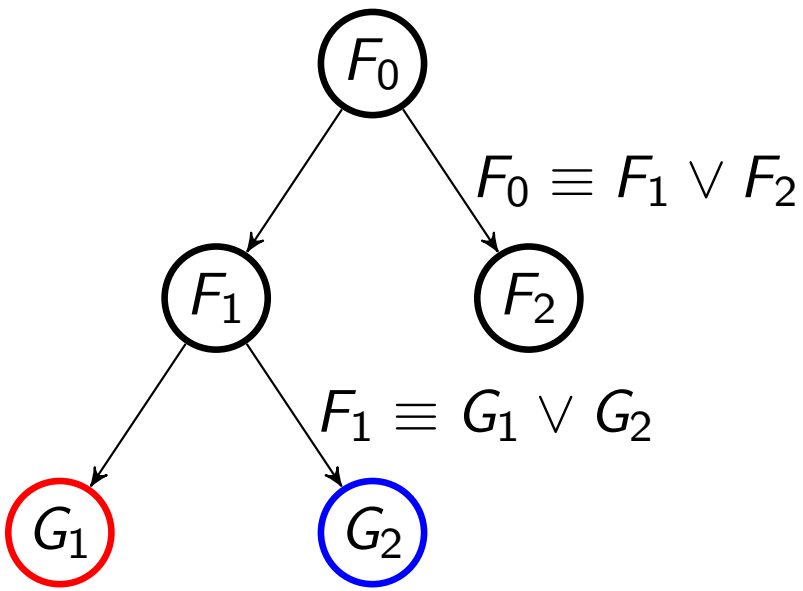
Search Tree

- A search tree whose root is labeled with the input formula
- A reduction rule transforms the formula F in the parent node into formulas G_i in the children nodes such that $F \equiv \bigvee G_i$
  - G_i is expected to be "simpler" than F
- G_0, G_1, G_2 are descendants of F_1 and F_0

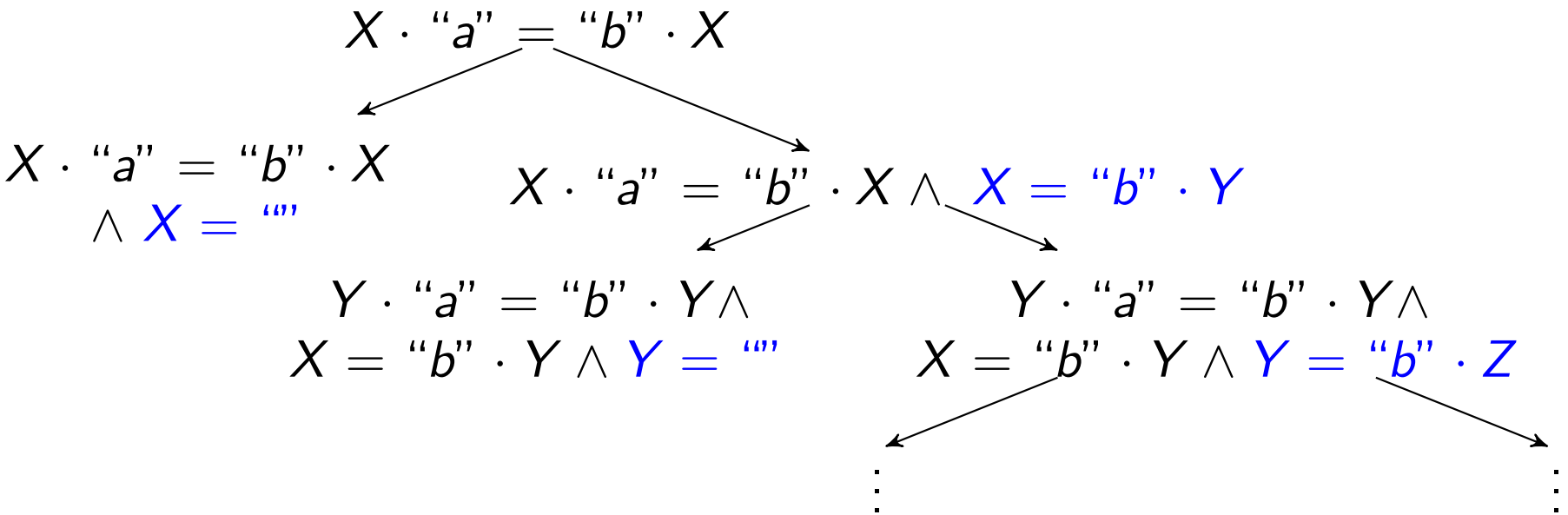


Search Strategy

- Breadth First search (BFS), Iterative Deepening search (IDS) are not practical
  - With depth $k \geq 0$: Have to traverse at least 2^k nodes.
- Depth First search (DFS) is more practical
  - Disadvantage: if G_1 is unsatisfiable and the subtree rooted at G_1 is infinite then DFS does not terminate



String Solving Example



- The reduced formula usually has more variables (e.g. Y, Z), and more constraints (e.g., $X = \text{“}b\text{”} \cdot Y$, $Y = \text{“}b\text{”} \cdot Z$)
- The length bounds of variables are likely changed (e.g., lower bound of length of X)
- Thus, syntactical checking is not able to detect loops
- Instead, we need to detect non-progression in solving process

Non-progression

- We propose a measure to know if the reduced formula is progressive towards a target solution
    - In our setting, we choose the minimal solution of the input formula
- If a formula labeling a node C does not contain the minimal solution of the input formula, then we will prune the subtree rooted at C
- Our reduction does not preserve the equivalence.
- Instead it preserves the minimal solution of the input formula

A measure for progression

- We define:
  - the lexical length of a solution w.r.t to some sequence.
    - If F has a solution and σ is a sequence of all the variables of F, then the lexical length of its minimal solution w.r.t. σ is denoted as $len(\sigma, F)$
    - If F has no solution then $len(\sigma, F)$ is $\top$
  - a total order for formulas: $F \preceq_\sigma G \stackrel{def}{=} len(\sigma, F) \leq len(\sigma, G)$

Progressive Pruning

- Let τ be a sequence of all variables of the input formula.
- A set of prunable subtrees of F is a set of its descendants G_i such that there exists a sequence σ of all variables of F satisfying:
  - τ is a prefix of σ and
  - $F \prec_\sigma G_i$ (the pruning condition check)
- We then *prune* derivation subtrees rooted at formulas G_i.

Theorems

- **Soundness**: Given an input formula F, if our algorithm
  - returns SAT: then F is satisfiable;
  - returns UNSAT: then F is unsatisfiable.

- **Semi-Completeness**: Suppose the given input formula F is satisfiable, and the pruning condition check is complete. Then our algorithm will
  - return SAT and
  - produce a minimal solution w.r.t. a sequence of all the variables of F.

  (A complete pruning condition check is non-trivial to implement)

Our string solver S3P

- Built on top of the string solver S3
  - String constraints include
    - Equality (e.g. $X = a$, where X is a variable and a is a constant)
    - Membership checking (e.g. $X \in r^\star$, where r is a regular expression)
    - String functions (e.g. **length**, **replace**)
  - Non-string constraints (e.g. integers)
- The pruning condition check
  - Sound but not complete
  - Practical (shown in Experiments)
- Conflict clause learning
  - Work in tandem with the pruning of non-progressive formulas
  - Practical (shown in Experiments)

Experiments

- Testing web applications by a dynamic symbolic execution (DSE) framework
- Comparison: There are 524 benchmarks that
  - S3 does not terminate
  - Z3-str2 detects overlapping variables and return UNKNOWN
  - S3P teminates and
    - return UNSAT for 215 benchmarks
    - return SAT for the remaining

	Norn	CVC4	S3	Z3-str2	S3P
Sat	27068	33227	34961	34931	35270
Unsat	11561	11625	11799	11799	12014
Unk	0	0	0	524	0
Error	6187	0	0	0	0
TO (20s)	2468	2432	524	30	0
Time (s)	178960	50346	16547	6309	6972

Table: Constraints generated by Kudzu

Experiments

- 102x faster
- Unsat cores are useful to speed up concolic testing/DSE and verification

# unsat files		12014
S3P	Time	1129s
S3P with unsat core	# unsat cores	59
	% skipped	99.5
	Time	11s

Table: Usefulness of unsatisfiable cores for Kudzu framework

Experiments

- Testing JavaScript web applications by Jalangi
- **replace** and sanitizers
- Z3-str2, CVC4, Norn do not support
- S3 does not terminate while S3P does

# benchmarks	# constraints	# **replace** operation	Time of S3P
48	624	96	143.7 s

Table: Constraints generated by Jalangi

Conclusion

- Progressive search strategy can detect loops caused by
  - unfolding recursive functions such as **replace**
  - unfolding **star** function when dealing with Kleene star operator
  - splitting string variables when dealing with string equation that involves overlapping variables
- Furthermore, we believe
  - it can work with a general fragment of equality logic
  - one possible direction is to apply for heap-allocated data structures such as linked lists
- Improve the robustness of S3P

Questions & Answers

Discussion

- Progressive Search
  - Search for one solution
  - Prune a search subtree if a shorter solution can be found elsewhere
  - Possibly prune away solutions

- Tableaux-based Search
  - Search for all solutions
  - Have pruning but never prune any solution

- Automatic Induction Proof Search (Chu *et al.* [2015])
  - Aim at proving entailment

N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321. Springer-Verlag, 2009.

J. R. Büchi and S. Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Mathematical Logic Quarterly*, pages 337–342, 1988.

Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. Automatic induction proofs of data-structures in imperative programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 457–466, New York, NY, USA, 2015. ACM.

M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *ACM-CCS*, pages 1232–1243. ACM, 2014.