Parse this!

Summoning Context-Sensitive Inputs with Goblin

Rob Lorch   Muhammad Daniyal Pirwani Dar
Cesare Tinelli   Omar Chowdhury

The University of Iowa
Stony Brook University

Problem Introduction

- Some software systems have highly complex input formats (e.g. compilers, file renderers, network protocol stacks)

Problem Introduction

- Some software systems have highly complex input formats (e.g. compilers, file renderers, network protocol stacks)
- Complex input formats are difficult for testing, especially automated testing

Problem Introduction

- Some software systems have highly complex input formats (e.g. compilers, file renderers, network protocol stacks)
- Complex input formats are difficult for testing, especially automated testing
- We will discuss techniques for automated input generation of software with complex input formats

Problem Introduction

- Some software systems have highly complex input formats (e.g. compilers, file renderers, network protocol stacks)
- Complex input formats are difficult for testing, especially automated testing
- We will discuss techniques for automated input generation of software with complex input formats
- Given an input specification, how to generate inputs?

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz
  - Can only handle context-free aspects of the spec

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz
  - Can only handle context-free aspects of the spec
  - Consider packet format with fields f_1 and f_2 s.t. $f_1 = |f_2|$

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz
  - Can only handle context-free aspects of the spec
  - Consider packet format with fields f_1 and f_2 s.t. $f_1 = |f_2|$
- Tools tailor-made for system under test (SUT)

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz
  - Can only handle context-free aspects of the spec
  - Consider packet format with fields f_1 and f_2 s.t. $f_1 = |f_2|$
- Tools tailor-made for system under test (SUT)
  - CSmith

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz
  - Can only handle context-free aspects of the spec
  - Consider packet format with fields f_1 and f_2 s.t. $f_1 = |f_2|$
- Tools tailor-made for system under test (SUT)
  - CSmith
  - Lacks generality!

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz
  - Can only handle context-free aspects of the spec
  - Consider packet format with fields f_1 and f_2 s.t. $f_1 = |f_2|$
- Tools tailor-made for system under test (SUT)
  - CSmith
  - Lacks generality!
- Coverage-guided mutation

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz
  - Can only handle context-free aspects of the spec
  - Consider packet format with fields f_1 and f_2 s.t. $f_1 = |f_2|$
- Tools tailor-made for system under test (SUT)
  - CSmith
  - Lacks generality!
- Coverage-guided mutation
  - How to generate an initial corpus?

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz
  - Can only handle context-free aspects of the spec
  - Consider packet format with fields f_1 and f_2 s.t. $f_1 = |f_2|$
- Tools tailor-made for system under test (SUT)
  - CSmith
  - Lacks generality!
- Coverage-guided mutation
  - How to generate an initial corpus?
  - What if no access to coverage information?

Existing Approaches

- Use context-free grammars (CFGs) to capture input structure
  - HTTP requests, DNS packets, LangFuzz
  - Can only handle context-free aspects of the spec
  - Consider packet format with fields f_1 and f_2 s.t. $f_1 = |f_2|$
- Tools tailor-made for system under test (SUT)
  - CSmith
  - Lacks generality!
- Coverage-guided mutation
  - How to generate an initial corpus?
  - What if no access to coverage information?
  - Mutations still blind to specification constraints

Wishlist

1. Generality of CFG-based approaches

Wishlist

1. Generality of CFG-based approaches
2. Expressiveness of tailor-made fuzzers

Wishlist

1. Generality of CFG-based approaches
2. Expressiveness of tailor-made fuzzers

Refine CFG-based approaches to be more expressive,

while maintaining their generality

Context-free input generation

First, a CFG-based input generation example: XML!

Context-free input generation

First, a CFG-based input generation example: XML!

```
<xml-tree> ::= <openclose-tag> |
               <open-tag> <inner-tree> <close-tag>

<inner-tree> ::= <TEXT> | <xml-tree>
               | <inner-tree> <inner-tree>

<open-tag>  ::= '<'  <id> '>' | '<' <id> ' ' <attribute> '>'
<close-tag> ::= '</' <id> '>'

<openclose-tag> := '<' <id> '/>'
                 | '<' <id> ' ' <attribute> '/>'

<attribute> ::= <id> '="' <TEXT > '"'
              | <attribute> ' ' <attribute>

<id> ::= <ID-START-CHAR> <ID-CHAR>*
```

Context-free input generation

Termination? (Mostly) well-formed inputs?

```
<xml-tree> ::= <openclose-tag> |
               <open-tag> <inner-tree> <close-tag>

<inner-tree> ::= <TEXT> | <xml-tree>
               | <inner-tree> <inner-tree>

<open-tag>  ::= ‘<’  <id> ‘>’ | ‘<’ <id> ‘ ’ <attribute> ‘>’
<close-tag> ::= ‘</’ <id> ‘>’

<openclose-tag> := ‘<’ <id> ‘/>’
                 | ‘<’ <id> ‘ ’ <attribute> ‘/>’

<attribute> ::= <id> ‘="’ <TEXT > ‘"’
              | <attribute> ‘ ’ <attribute>

<id> ::= <ID-START-CHAR> <ID-CHAR>*
```

Introducing Goblin

- Presenting Goblin

Introducing Goblin

- Presenting Goblin
  - A context-sensitive input generation tool, supported by

Introducing GOBLIN

- Presenting GOBLIN
  - A context-sensitive input generation tool, supported by
  - A new DSL with a formal semantics

Introducing Goblin

- Presenting Goblin
  - A context-sensitive input generation tool, supported by
  - A new DSL with a formal semantics
  - A new search algorithm to address the underlying problem

Introducing GOBLIN

- Presenting GOBLIN
  - A context-sensitive input generation tool, supported by
  - A new DSL with a formal semantics
  - A new search algorithm to address the underlying problem
  - Supporting constraint solving over arbitrary SMT theories

Introducing GOBLIN

- Presenting GOBLIN
  - A context-sensitive input generation tool, supported by
  - A new DSL with a formal semantics
  - A new search algorithm to address the underlying problem
  - Supporting constraint solving over arbitrary SMT theories
  - And evaluated in the context of real fuzzing workflows

Introducing Goblin

- Presenting Goblin
  - A context-sensitive input generation tool, supported by
  - A new DSL with a formal semantics
  - A new search algorithm to address the underlying problem
  - Supporting constraint solving over arbitrary SMT theories
  - And evaluated in the context of real fuzzing workflows
- Given a context-sensitive grammar G, find x such that $x \in \mathcal{L}(G)$

Introducing GOBLIN

- Presenting GOBLIN
  - A context-sensitive input generation tool, supported by
  - A new DSL with a formal semantics
  - A new search algorithm to address the underlying problem
  - Supporting constraint solving over arbitrary SMT theories
  - And evaluated in the context of real fuzzing workflows
- Given a context-sensitive grammar G, find x such that $x \in \mathcal{L}(G)$
- What is a context-sensitive grammar?

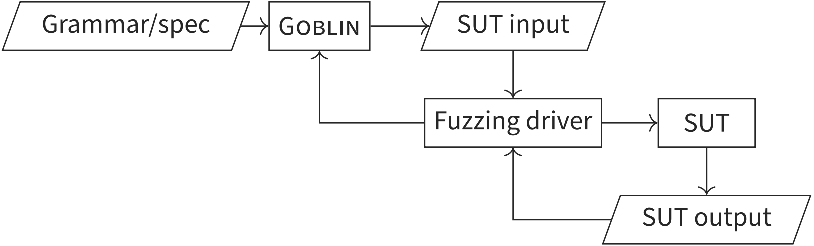
```
<xml-tree> ::= <openclose-tag> |
               <open-tag> <inner-tree> <close-tag>
             { <open-tag>.<id> = <close-tag>.<id> }
...
```

Input generation vs fuzzing

GOBLIN is an input generator, but not a (complete) fuzzer

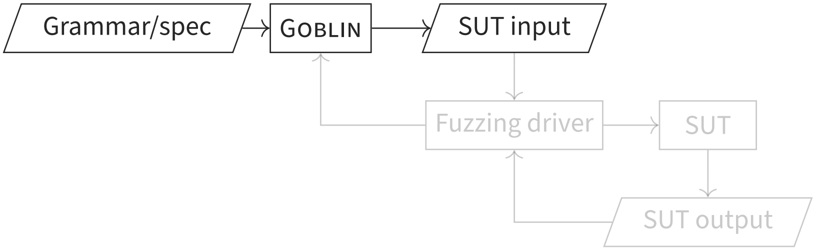
Input generation vs fuzzing

GOBLIN is an input generator, but not a (complete) fuzzer



Input generation vs fuzzing

GOBLIN is an input generator, but not a (complete) fuzzer



From now on we only discuss input generation

GOBLIN by example: Language features

GOBLIN grammars have production rules as in CFGs

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>;



<PAYLOAD> ::= <F1> <F2> <BYTES>;

<BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>;
```

Goblin by example: Language features

Use symbolic terminals (in teal) with type annotations rather than concrete terminals. Capture abstract syntax, not concrete syntax

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>;



<PAYLOAD> ::= <F1> <F2> <BYTES>;

<BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>;
<TYPE> :: BitVec(8);
<BYTE> :: BitVec(8);
<AUX> :: BitVec(8);
<F1> :: BitVec(8); <F2> :: BitVec(8); <OPT> :: BitVec(4);
```

Goblin by example: Language features

Constrain symbolic terminals with refinement types

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>;



<PAYLOAD> ::= <F1> <F2> <BYTES>;

<BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>;
<TYPE> :: BitVec(8) { <TYPE> = 0x01 or <TYPE> = 0x02; };
<BYTE> :: BitVec(8) { <BYTE> bvult 0x88;};
<AUX> :: BitVec(8);
<F1> :: BitVec(8); <F2> :: BitVec(8); <OPT> :: BitVec(4);
```

GOBLIN by example: Language features

Attach semantic constraits to production rules

Support types/functions/predicates with SMT-LIB analogues

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>
  { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
    <TYPE> = 0x01 => (<PAYLOAD>.<BYTES>.<BYTE> bvugt 0x20
                and <PAYLOAD>.<BYTES>.<BYTE> bvult 0x7E); };
<PAYLOAD> ::= <F1> <F2> <BYTES>
 { <BYTES>.<OPT> bvugt 0x0; };
<BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>;
<TYPE> :: BitVec(8) { <TYPE> = 0x01 or <TYPE> = 0x02; };
<BYTE> :: BitVec(8) { <BYTE> bvult 0x88;};
<AUX> :: BitVec(8);
<F1> :: BitVec(8); <F2> :: BitVec(8); <OPT> :: BitVec(4);
```

GOBLIN by example: Language features

Reference child nonterminals with dot notation

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>
  { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
    <TYPE> = 0x01 => (<PAYLOAD>.<BYTES>.<BYTE> bvugt 0x20
                and <PAYLOAD>.<BYTES>.<BYTE> bvult 0x7E); };
<PAYLOAD> ::= <F1> <F2> <BYTES>
 { <BYTES>.<OPT> bvugt 0x0; };
<BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>;
<TYPE> :: BitVec(8) { <TYPE> = 0x01 or <TYPE> = 0x02; };
<BYTE> :: BitVec(8) { <BYTE> bvult 0x88;};
<AUX> :: BitVec(8);
<F1> :: BitVec(8); <F2> :: BitVec(8); <OPT> :: BitVec(4);
```

Goblin by example: Language features

Dot notation is partial and implicitly universally quantified

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>
  { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
    <TYPE> = 0x01 => (<PAYLOAD>.<BYTES>.<BYTE> bvugt 0x20
                        and <PAYLOAD>.<BYTES>.<BYTE> bvult 0x7E); };
<PAYLOAD> ::= <F1> <F2> <BYTES>
 { <BYTES>.<OPT> bvugt 0x0; };
<BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>;
<TYPE> :: BitVec(8) { <TYPE> = 0x01 or <TYPE> = 0x02; };
<BYTE> :: BitVec(8) { <BYTE> bvult 0x88;};
<AUX> :: BitVec(8);
<F1> :: BitVec(8); <F2> :: BitVec(8); <OPT> :: BitVec(4);
```

Goblin by example: Language features

Many constraints are not amenable to automated constraint solving with an SMT engine

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>
  { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
...
<PAYLOAD> ::= <F1> <F2> <BYTES>
...
```

GOBLIN by example: Language features

Many constraints are not amenable to automated constraint solving with an SMT engine

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>
  { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
...
<PAYLOAD> ::= <F1> <F2> <BYTES>
...
```

Derived fields with `<-` denote nonterminals that are directly computable, enforced syntactically

Goblin by example: Language features

Many constraints are not amenable to automated constraint solving with an SMT engine

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>
  { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
...
<PAYLOAD> ::= <F1> <F2> <BYTES>
...
```

Derived fields with `<-` denote nonterminals that are directly computable, enforced syntactically

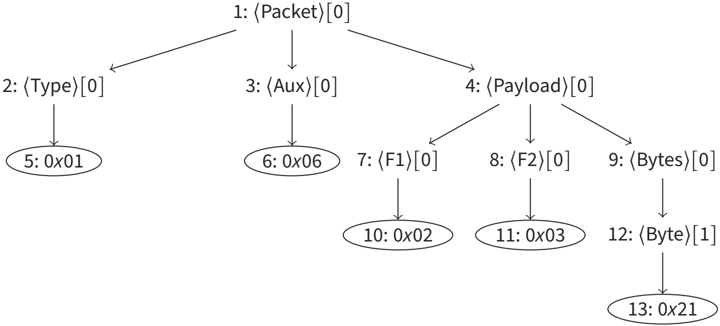
Cryptographic hashes, checksums, or any computable function

Semantics: Abstract syntax trees

GOBLIN generates abstract syntax trees, not strings

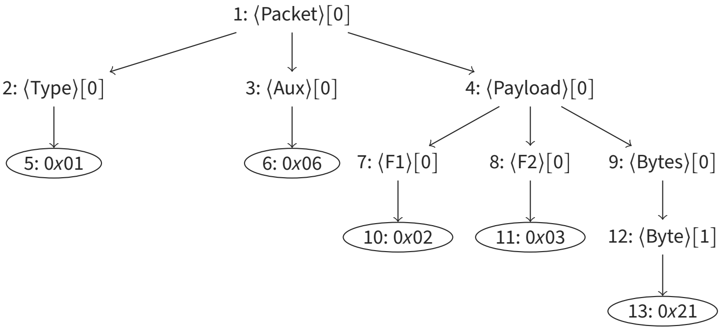
Semantics: Abstract syntax trees

Goblin generates abstract syntax trees, not strings



Semantics: Abstract syntax trees

GOBLIN generates abstract syntax trees, not strings



Not a string `0x0106020321`

Semantics: Abstract syntax trees

- Without the AST view, prior work natively only supports string constraints

Semantics: Abstract syntax trees

- Without the AST view, prior work natively only supports string constraints
- AST semantics position GOBLIN as a generator of constrained algebraic datatype (ADT) terms

Semantics: Abstract syntax trees

- Without the AST view, prior work natively only supports string constraints
- AST semantics position GOBLIN as a generator of constrained algebraic datatype (ADT) terms

The ADT view allows GOBLIN to natively support constraints over arbitrary SMT theories

Semantics

The semantics of a goblin input G are the set of ASTs that respect (*i*) G's context-free syntactic constraints and G's (*ii*) context-sensitive semantic constraints

Semantics

The semantics of a goblin input G are the set of ASTs that respect (i) G's context-free syntactic constraints and G's (ii) context-sensitive semantic constraints

$\mathcal{L}_{\mathsf{AST}}(G) = \{t \mid$

1. t is rooted at the start symbol

Semantics

The semantics of a goblin input G are the set of ASTs that respect (i) G's context-free syntactic constraints and G's (ii) context-sensitive semantic constraints

$\mathcal{L}_{\mathsf{AST}}(G) = \{t \mid$

1. t is rooted at the start symbol
2. Each $v \in V(t)$ is either

Semantics

The semantics of a goblin input G are the set of ASTs that respect (i) G's context-free syntactic constraints and G's (ii) context-sensitive semantic constraints

$\mathcal{L}_{\text{AST}}(G) = \{t \mid$

1. t is rooted at the start symbol
2. Each $v \in V(t)$ is either
    a. A non-leaf with children representing a production rule application in G,

Semantics

The semantics of a goblin input G are the set of ASTs that respect (i) G's context-free syntactic constraints and G's (ii) context-sensitive semantic constraints

$\mathcal{L}_{\text{AST}}(G) = \{t \mid$

1. t is rooted at the start symbol
2. Each $v \in V(t)$ is either
    a. A non-leaf with children representing a production rule application in G,
    b. A non-leaf representing a type annotation in G, or

Semantics

The semantics of a goblin input G are the set of ASTs that respect (i) G's context-free syntactic constraints and G's (ii) context-sensitive semantic constraints

$\mathcal{L}_{\text{AST}}(G) = \{t \mid$

1. t is rooted at the start symbol
2. Each $v \in V(t)$ is either
    a. A non-leaf with children representing a production rule application in G,
    b. A non-leaf representing a type annotation in G, or
    c. A well-typed leaf

Semantics

The semantics of a goblin input G are the set of ASTs that respect (i) G's context-free syntactic constraints and G's (ii) context-sensitive semantic constraints

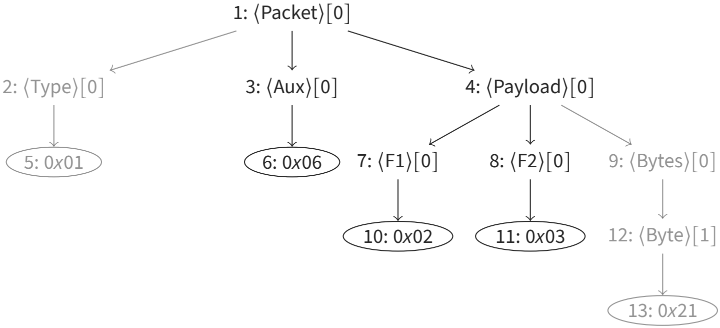
$\mathcal{L}_{\text{AST}}(G) = \{t \mid$

1. t is rooted at the start symbol
2. Each $v \in V(t)$ is either
    a. A non-leaf with children representing a production rule application in G,
    b. A non-leaf representing a type annotation in G, or
    c. A well-typed leaf
3. t satisfies the constraints at every production rule application

$\}$

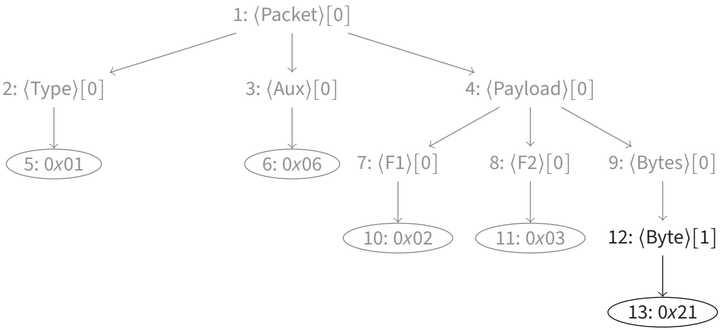
Semantics: Constraint Satisfaction

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>
  { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
...
<PAYLOAD> ::= <F1> <F2> <BYTES>
...
```



Semantics: Constraint Satisfaction

```
...
<BYTE> :: BitVec(8) { <BYTE> bvult 0x88; };
...
```



GOBLIN Workflow

How does GOBLIN find t such that $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

GOBLIN Workflow

How does GOBLIN find t such that $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

General workflow:

Goblin Workflow

How does Goblin find t such that $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

General workflow:

1. Context-free wellformedness and other syntactic checks

Goblin Workflow

How does Goblin find t such that $t \in \mathcal{L}_{\mathrm{AST}}(G)$?

General workflow:

1. Context-free wellformedness and other syntactic checks
2. Type checking

GOBLIN Workflow

How does GOBLIN find t such that $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

General workflow:

1. Context-free wellformedness and other syntactic checks
2. Type checking
3. Desugar refinement types

GOBLIN Workflow

How does GOBLIN find t such that $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

General workflow:

1. Context-free wellformedness and other syntactic checks
2. Type checking
3. Desugar refinement types
4. Disambiguate constraints

Goblin Workflow

How does Goblin find t such that $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

General workflow:

1. Context-free wellformedness and other syntactic checks
2. Type checking
3. Desugar refinement types
4. Disambiguate constraints
5. Abstract derived fields

GOBLIN Workflow

How does GOBLIN find t such that $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

General workflow:

1. Context-free wellformedness and other syntactic checks
2. Type checking
3. Desugar refinement types
4. Disambiguate constraints
5. Abstract derived fields
6. Main search algorithm

Goblin Workflow

How does Goblin find t such that $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

General workflow:

1. Context-free wellformedness and other syntactic checks
2. Type checking
3. Desugar refinement types
4. Disambiguate constraints
5. Abstract derived fields
6. Main search algorithm
7. Compute derived fields

GOBLIN Workflow

How does GOBLIN find t such that $t \in \mathcal{L}_{\mathrm{AST}}(G)$?

General workflow:

1. Context-free wellformedness and other syntactic checks
2. Type checking
3. Desugar refinement types
4. Disambiguate constraints
5. Abstract derived fields
6. Main search algorithm
7. Compute derived fields
8. Serialize

Context-Free Wellformedness

- Context-free language emptiness (e.g., $S \rightarrow S$) is a modeling issue, stemming from non-wellfounded recursion

Context-Free Wellformedness

- Context-free language emptiness (e.g., $S \rightarrow S$) is a modeling issue, stemming from non-wellfounded recursion
- We perform a more general check called context-free wellformedness

Context-Free Wellformedness

- Context-free language emptiness (e.g., $S \to S$) is a modeling issue, stemming from non-wellfounded recursion
- We perform a more general check called context-free wellformedness

1. Iteratively expand a set N of nonterminals known to be able to produce finite derivations until reaching a fixpoint

Context-Free Wellformedness

- Context-free language emptiness (e.g., $S \to S$) is a modeling issue, stemming from non-wellfounded recursion
- We perform a more general check called context-free wellformedness

1. Iteratively expand a set N of nonterminals known to be able to produce finite derivations until reaching a fixpoint
2. Check every reachable nonterminal is in N

Context-Free Wellformedness

- Context-free language emptiness (e.g., $S \rightarrow S$) is a modeling issue, stemming from non-wellfounded recursion
- We perform a more general check called context-free wellformedness

1. Iteratively expand a set N of nonterminals known to be able to produce finite derivations until reaching a fixpoint
2. Check every reachable nonterminal is in N

```
<SAE_PACKET> ::= <COMMIT> | <CONFIRM>;
<COMMIT> ::= <FIELD> <RG_ID_LIST>;
<RG_ID_LIST> ::= <RG_ID> <RG_ID_LIST>;
<CONFIRM> ::= <FIELD1> <FIELD2>;
...
```

Derived Field Checks

Derived fields $\langle F \rangle \leftarrow e$ must be computable without constraint solving

Derived Field Checks

Derived fields $\langle F \rangle \leftarrow e$ must be computable without constraint solving

1. Disallow cyclic dependencies (e.g. $\langle A \rangle \rightarrow T[\langle B \rangle], \langle B \rangle \rightarrow T[\langle A \rangle]$)

Derived Field Checks

Derived fields $\langle F \rangle \leftarrow e$ must be computable without constraint solving

1. Disallow cyclic dependencies (e.g. $\langle A \rangle \rightarrow T[\langle B \rangle], \langle B \rangle \rightarrow T[\langle A \rangle]$)
    - (i) Build a directed graph for each prod rule

Derived Field Checks

Derived fields $\langle F \rangle \leftarrow e$ must be computable without constraint solving

1. Disallow cyclic dependencies (e.g. $\langle A \rangle \rightarrow T[\langle B \rangle], \langle B \rangle \rightarrow T[\langle A \rangle]$)
   - (i) Build a directed graph for each prod rule
   - (ii) Include vertex for each RHS nonterminal/symbolic terminal

Derived Field Checks

Derived fields $\langle F \rangle \leftarrow e$ must be computable without constraint solving

1. Disallow cyclic dependencies (e.g. $\langle A \rangle \rightarrow T[\langle B \rangle], \langle B \rangle \rightarrow T[\langle A \rangle]$)
    - (i) Build a directed graph for each prod rule
    - (ii) Include vertex for each RHS nonterminal/symbolic terminal
    - (iii) Check for cycles

Derived Field Checks

Derived fields $\langle F \rangle \leftarrow e$ must be computable without constraint solving

1. Disallow cyclic dependencies (e.g. $\langle A \rangle \rightarrow T[\langle B \rangle], \langle B \rangle \rightarrow T[\langle A \rangle]$)
   (*i*) Build a directed graph for each prod rule
   (*ii*) Include vertex for each RHS nonterminal/symbolic terminal
   (*iii*) Check for cycles
2. Disallow derived fields in semantic constraints

Derived Field Checks

Derived fields $\langle F \rangle \leftarrow e$ must be computable without constraint solving

1. Disallow cyclic dependencies (e.g. $\langle A \rangle \rightarrow T[\langle B \rangle], \langle B \rangle \rightarrow T[\langle A \rangle]$)
   - (*i*) Build a directed graph for each prod rule
   - (*ii*) Include vertex for each RHS nonterminal/symbolic terminal
   - (*iii*) Check for cycles
2. Disallow derived fields in semantic constraints
   - E.g., $\langle D \rangle > 0$ where $\langle D \rangle$ is derived

Abstract derived fields

We remove derived fields from G before the main search; compute them after constraint solving

Abstract derived fields

We remove derived fields from *G* before the main search; compute them after constraint solving

```
<PACKET> ::= <TYPE> <AUX> <PAYLOAD>
  { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
...
```

$\rightsquigarrow$

```
<PACKET> ::= <TYPE> dep_sym_leaf <PAYLOAD>
  { ...
```

Search algorithm

Formal guarantees

In the paper, we formalize the Goblin search procedure as a calculus comprised of 11 inference rules

Formal guarantees

In the paper, we formalize the GOBLIN search procedure as a calculus comprised of 11 inference rules

We prove the calculus satisfies solution soundness, refutation soundness, and solution completeness

Evaluation

Case study 1, 2, 3: Compare directly with prior work, ISLa

Evaluation

Case study 1, 2, 3: Compare directly with prior work, ISLa

- XML, ScriptSize C, and CSV input generation

Evaluation

Case study 1, 2, 3: Compare directly with prior work, ISLa

- XML, ScriptSize C, and CSV input generation
- Matching tags, definition before use, and column count

Evaluation

Case study 1, 2, 3: Compare directly with prior work, ISLa

- XML, ScriptSize C, and CSV input generation
- Matching tags, definition before use, and column count
- Measure efficiency in inputs generated per minute and diversity in k-path coverage for $k = 3$ (percentage of paths of length 3 traversed through the grammar)

Evaluation

Case study 1, 2, 3: Compare directly with prior work, ISLa

- XML, ScriptSize C, and CSV input generation
- Matching tags, definition before use, and column count
- Measure efficiency in inputs generated per minute and diversity in k-path coverage for $k = 3$ (percentage of paths of length 3 traversed through the grammar)
- Outperform prior work by $\sim 10 - 100\%$ in all metrics, save for efficiency of CSV inputs

Evaluation

Case study 4

Evaluation

Case study 4

- WiFi SAE packet input generation (SAECRED's SyGuS engine; GOBLIN predecessor)

Evaluation

Case study 4

- WiFi SAE packet input generation (SAECRED's SyGuS engine; GOBLIN predecessor)
- Handle bit vector SMT constraints not expressible in ISLa

Evaluation

Case study 4

- WiFi SAE packet input generation (SAECRED's SyGuS engine; GOBLIN predecessor)
- Handle bit vector SMT constraints not expressible in ISLa
- 36x improvement in efficiency

Evaluation

Case study 4

- WiFi SAE packet input generation (SAECRED's SyGuS engine; Goblin predecessor)
- Handle bit vector SMT constraints not expressible in ISLa
- 36x improvement in efficiency
- Able to produce outputs for more than twice the number of grammars (~24,000 / ~97000 up to ~49,000 / ~97,000)

Discussion and future work

- User-facing language
  - Synthesized and inherited attributes
  - User-defined recursive functions
  - Built-ins like `length(.)`

Discussion and future work

- User-facing language
  - Synthesized and inherited attributes
  - User-defined recursive functions
  - Built-ins like `length(.)`
- More powerful type system (e.g., polymorphism)

Discussion and future work

- User-facing language
  - Synthesized and inherited attributes
  - User-defined recursive functions
  - Built-ins like `length(.)`
- More powerful type system (e.g., polymorphism)
- AI abstraction/refinement algorithms

Discussion and future work

- User-facing language
  - Synthesized and inherited attributes
  - User-defined recursive functions
  - Built-ins like `length(.)`
- More powerful type system (e.g., polymorphism)
- AI abstraction/refinement algorithms
- Finite model finding and CLP engines

Discussion and future work

- User-facing language
  - Synthesized and inherited attributes
  - User-defined recursive functions
  - Built-ins like `length(.)`
- More powerful type system (e.g., polymorphism)
- AI abstraction/refinement algorithms
- Finite model finding and CLP engines
- Divide and conquer/parallel approaches

Discussion and future work

- User-facing language
  - Synthesized and inherited attributes
  - User-defined recursive functions
  - Built-ins like `length(.)`
- More powerful type system (e.g., polymorphism)
- AI abstraction/refinement algorithms
- Finite model finding and CLP engines
- Divide and conquer/parallel approaches
- CDCL-style backjumping

Thanks! Questions?

robert-lorch@uiowa.edu

`github.com/lorchrob/goblin`

Structural constraints

Every element of the second list is present in the first

```
<S> ::= <L1> <L2>
      { <L2>.<_defs_down> = <L1>.<_defs_up>; };
<L1> ::= <_defs_up> <str> <L1>
       { <_defs_up> = set.union(set.singleton(<str>),
                                <L1>.<_defs_up>); }
       | <_defs_up> <str>
       { <_defs_up> = set.singleton(<str>); };
<L2> ::= <_defs_down> <str> <L2>
       { set.member(<str>, <_defs_down>);
         <L2>.<_defs_down> = <_defs_down>; }
       | <_defs_down> <str>
       { set.member(<str>, <_defs_down>); };
<str> :: String;
<_defs_down> :: Set(String);
<_defs_up> :: Set(String);
```

Prior Work

- ISLa, most similar in spirit
  - Only natively supports string constraints
  - Global constraints not amenable to grammar mutations
- Fandango
  - Uses genetic algorithms with built-in fitness functions
  - Constraints may be non-monotonic (no monotonically decreasing notion of distance for constraint satisfaction)
  - Times out for simple examples (equality, set membership)
  - Times out with SAECRED grammars

Related Work

- Parser generator libraries (e.g. ANTLR, yacc) handle context sensitivity but are for parsing rather than generation
- Attribute grammars handle context sensitivity but work focuses on parsing and theoretical results
- Property-based testing does not support general context-sensitive constraints over inputs
- SyGuS does not natively support constraints over non-top-level nonterminals

Local constraints for grammar mutation

- Crossover mutation swapping `GROUP_ID` and `RG_LIST`
- Local constraints are easier to maintain

```
COMMIT ::= SEQ GROUP_ID SCALAR
  { GROUP_ID is equal to 13  and  SEQ is greater than 4 }
RG_CONT ::= RG_LENGTH RG_TY RG_LIST
  { RG_LENGTH is the length of RG_LIST }
...
⇝
COMMIT ::= SEQ RG_LIST SCALAR
  { SEQ is greater than 4 }
RG_CONT ::= RG_LENGTH RG_TY GROUP_ID
  { GROUP_ID is equal to 13 }
...
```

Semantics: Interpretation function

Interpretation function $\mathcal{I}_{tr}(t)$ outputs denotation of term t in AST tr

$\mathcal{I}_{tr}$ also maps function and predicate symbols to their fixed interpretations

$\mathcal{I}_{tr}(f(t_1, \ldots, t_n)) = \top$ if some $\mathcal{I}_{tr}(t_i) = \top$

$\mathcal{I}_{tr}(f(t_1, \ldots, t_n)) = \mathcal{I}_{tr}(f)(\mathcal{I}_{tr}(t_1), \ldots, \mathcal{I}_{tr}(t_n))$

$I_{tr}(\langle \mathrm{nt} \rangle[i].\langle \mathrm{nt\_expr} \rangle) = I_{tr'}(\langle \mathrm{nt\_expr} \rangle)$ for tr' rooted at the only

$v \in \mathrm{get\_children}(\mathrm{tr}, \mathrm{root}(\mathrm{tr}))$ such that $\ell(v) = \langle \mathrm{nt} \rangle[i]$

$\ldots$

Semantics: Satisfaction relation

Satisfaction relation $\models_{\mathcal{G}}$ captures whether or not a given constraint in G is satisfied by a given AST

$\models_G \varphi$ if $\mathcal{I}_{tr}(t_i) = \top$ for some subterm t_i of φ; otherwise,

$\models_{\mathcal{G}} p(t_1, \ldots, t_n)$ if $(\mathcal{I}_{tr}(t_1), \ldots, \mathcal{I}_{tr}(t_n)) \in \mathcal{I}_{tr}(p)$

$\models_{\mathcal{G}} \neg\varphi$ if $\not\models_{\mathcal{G}} \varphi$

$tr \models_{\mathcal{G}} \varphi_1 \wedge \varphi_2$ if $tr \models_{\mathcal{G}} \varphi_1$ and $tr \models_{\mathcal{G}} \varphi_2$

$tr \models_{\mathcal{G}} \varphi_1 \vee \varphi_2$ if $tr \models_{\mathcal{G}} \varphi_1$ or $tr \models_{\mathcal{G}} \varphi_2$

$tr \models_{\mathcal{G}} \varphi_1 \Rightarrow \varphi_2$ if $tr \not\models_G \varphi_1$ or $tr \models_{\mathcal{G}} \varphi_2$

Semantics: Denotation of G

Semantics of a GOBLIN input G is the set of syntactically valid ASTs which satisfy all the constraints

$$\begin{aligned}\llbracket G \rrbracket = \{t \mid\ & t \in \mathcal{L}_{\mathsf{AST}}(G) \wedge \forall(\mathsf{nt}, \_, \mathsf{constraints}) \in R.\\ & \forall s \in \mathsf{get\_subtrees}(t, \mathsf{nt}).\ \forall \varphi \in \mathsf{constraints}.\ s \vDash_{\mathcal{G}} \mathsf{resolve}(\varphi)\}\end{aligned}$$

Calculus

Can conceptualize as guarded rewrite rules on global state

Example rule expands a symbolic terminal

$$\textsc{NormalizeTA}\ \frac{v \in \text{open\_leaves}(DT) \quad \text{depth}(DT) \le L \quad \ell(v), \tau \in \Gamma}{DT' \leftarrow \text{expand}_G(DT, v, [])}$$

Search algorithm pseudocode

```
initializeGlobalState()
while ¬allLeavesClosed(dt) do
    if there is more than one unvisited expansion then
        Decide
    else
        Propagate
    while ¬is_normalized(dt) do
        NormalizePr (if applicable)
        NormalizeTa (if applicable)
    if current search depth d > depth limit L then
        if assertionLevel = 1 then
            RestartDepth
        else
            BacktrackDepth
    else
        for all c ∈ constraint set C do
            Assert if c is applicable
        if smt_check_sat() = UNSAT then
            if assertionLevel = 1 ∧ ¬bd? then
                Fail
            else if assertionLevel = 1 then
                RestartUnsat
            else
                BacktrackUnsat
```