# 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?

# Roadmap

1. Existing approaches to input generation
2. Syntax and language features
3. Semantics
4. GOBLIN workflow
   - Well-formedness checks
   - Derived fields
   - Main search algorithm
5. Formal guarantees
6. Evaluation
7. Discussion and future work

1. **Existing approaches to input generation**
2. Syntax and language features
3. Semantics
4. GOBLIN workflow
   - Well-formedness checks
   - Derived fields
   - Main search algorithm
5. Formal guarantees
6. Evaluation
7. Discussion and future work

# 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

An input generation approach that gives...

# Wishlist

An input generation approach that gives...

1. Generality of CFG-based approaches

# Wishlist

An input generation approach that gives...

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

# Wishlist

An input generation approach that gives...

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? See ANTLR, yacc

# 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? See ANTLR, yacc

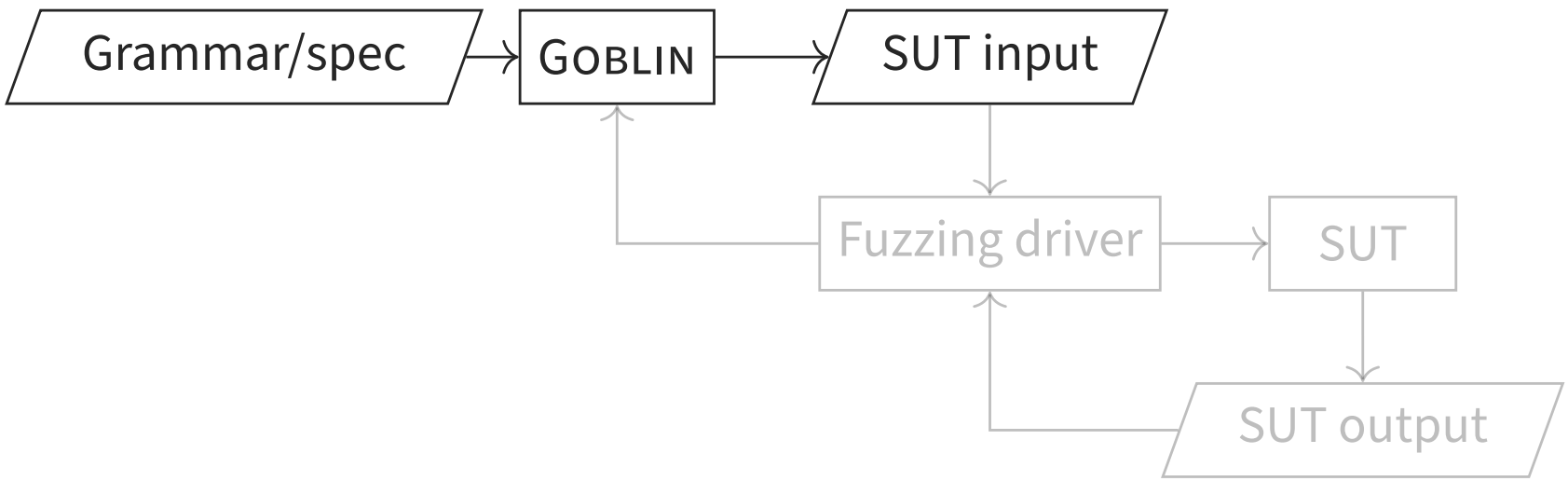
```
1 <xml-tree> ::= <openclose-tag> |
2                <open-tag> <inner-tree> <close-tag>
3              { <open-tag>.<id> = <close-tag>.<id> }
4 ...
```

# Input generation vs fuzzing

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

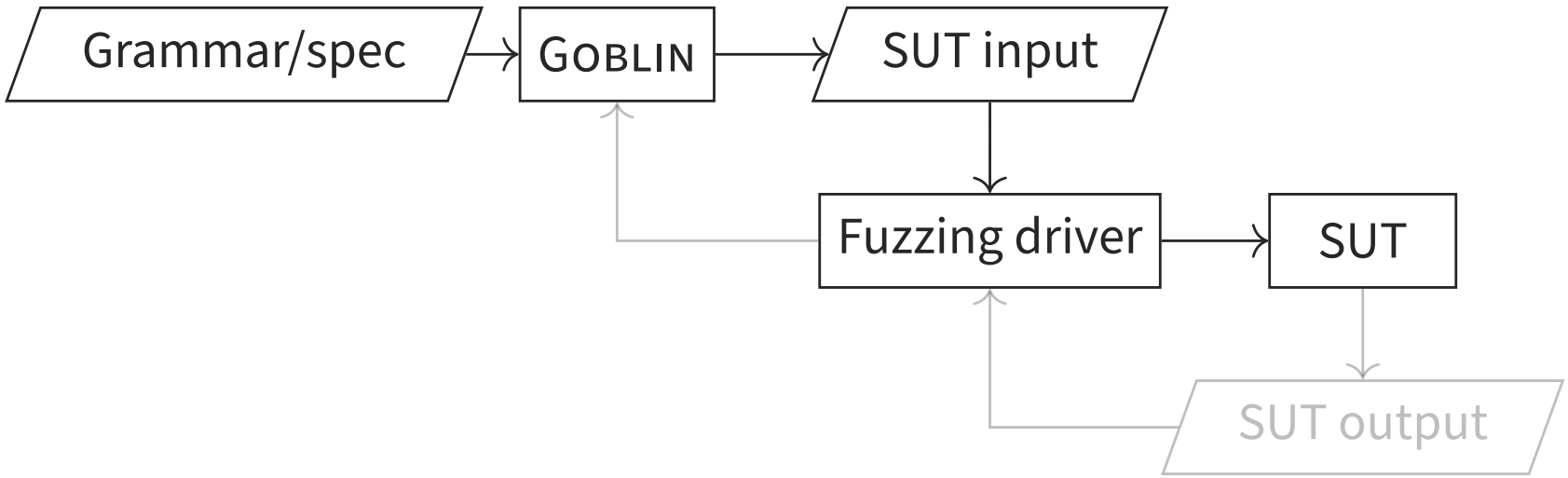
# Input generation vs fuzzing

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



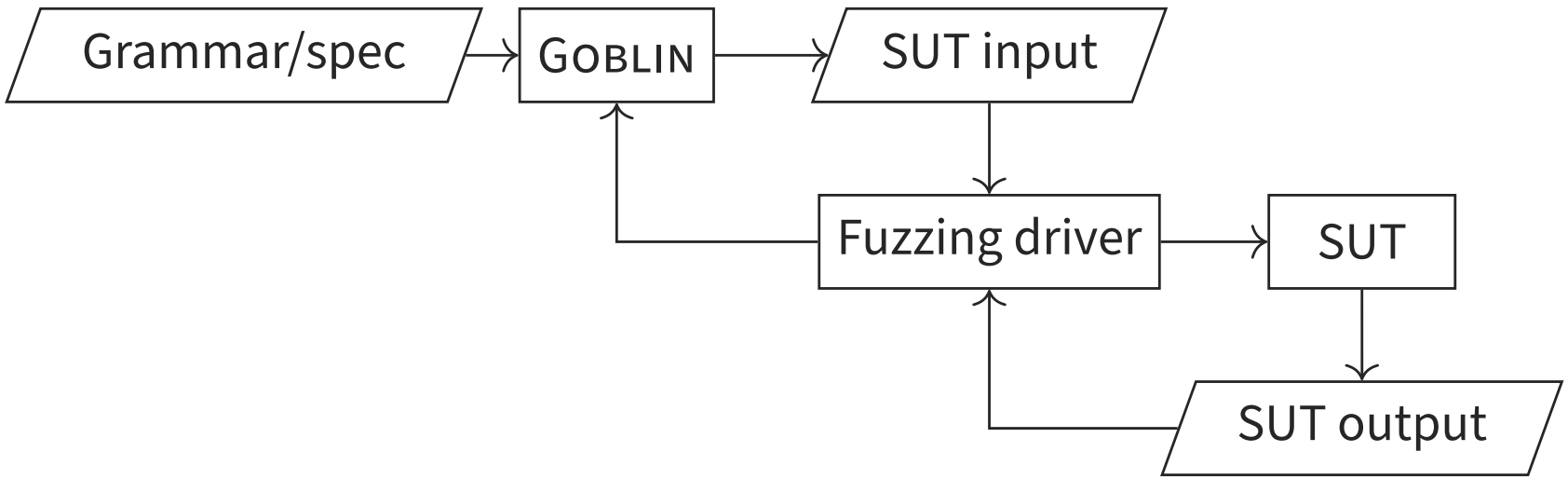
# Input generation vs fuzzing

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



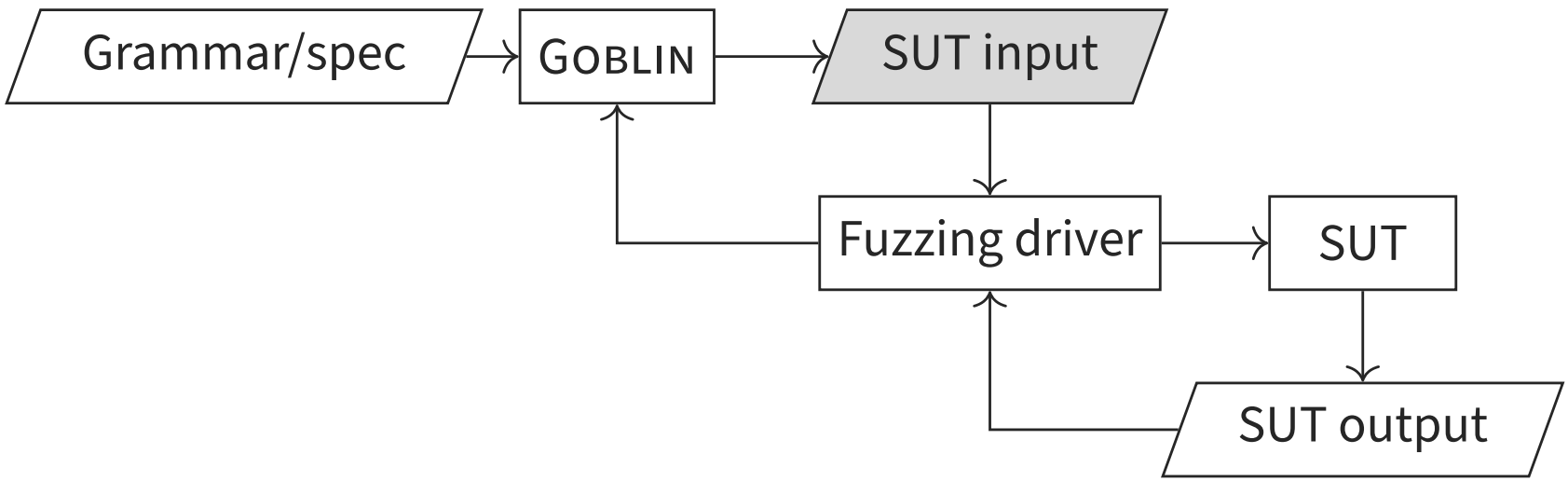
# Input generation vs fuzzing

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



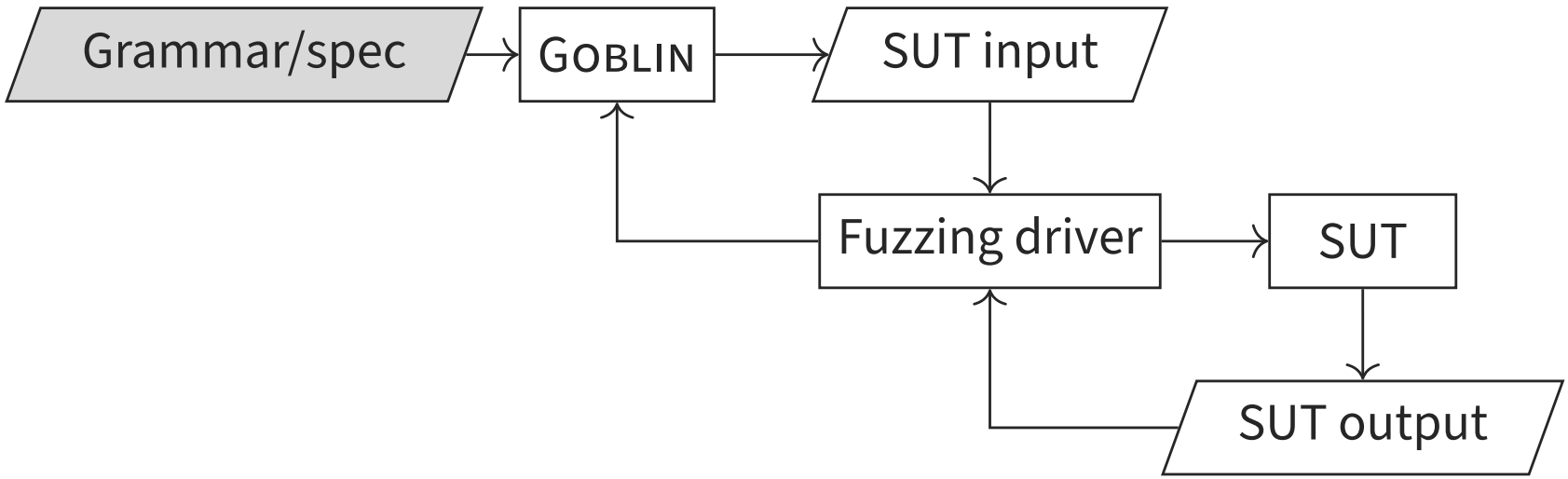
# Input generation vs fuzzing

Most tools perform mutations at the byte level



# Input generation vs fuzzing

Some tools perform structure-aware mutations (ATFuzz, SAECRED)



# Structure-aware mutations

How to perform context-sensitive, structure-aware mutations?

# Structure-aware mutations

How to perform context-sensitive, structure-aware mutations?

Crossover mutation swapping `GROUP_ID` and `RG_LIST`

```
1  COMMIT ::= SEQ GROUP_ID SCALAR
2    { GROUP_ID is equal to 13  and  SEQ is greater than 4 }
3  RG_CONT ::= RG_LENGTH RG_TY RG_LIST
4    { RG_LENGTH is the length of RG_LIST }
5  ...
6  ⇝
7  COMMIT ::= SEQ RG_LIST SCALAR
8    { SEQ is greater than 4 }
9  RG_CONT ::= RG_LENGTH RG_TY GROUP_ID
10   { GROUP_ID is equal to 13 }
11 ...
```

# Structure-aware mutations

How to perform context-sensitive, structure-aware mutations?

Crossover mutation swapping `GROUP_ID` and `RG_LIST`

```
1  COMMIT ::= SEQ GROUP_ID SCALAR
2    { GROUP_ID is equal to 13  and  SEQ is greater than 4 }
3  RG_CONT ::= RG_LENGTH RG_TY RG_LIST
4    { RG_LENGTH is the length of RG_LIST }
5  ...
6  ⇝
7  COMMIT ::= SEQ RG_LIST SCALAR
8    { SEQ is greater than 4 }
9  RG_CONT ::= RG_LENGTH RG_TY GROUP_ID
10   { GROUP_ID is equal to 13 }
11 ...
```

**Prior work cannot tie global constraints to mutations**

# Prior work

# Prior work

ISLa

- Global constraints make structure-aware mutations less precise
- Only natively supports string constraints
- “Maximalist” design philosophy

# Prior work

ISLa

- Global constraints make structure-aware mutations less precise
- Only natively supports string constraints
- “Maximalist” design philosophy

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)

1. Existing approaches to input generation
2. **Syntax and language features**
3. Semantics
4. GOBLIN workflow
   - Well-formedness checks
   - Derived fields
   - Main search algorithm
5. Formal guarantees
6. Evaluation
7. Discussion and future work

# GOBLIN by example: Language features

GOBLIN's minimalist design philosophy positions it as an intermediate language (compile target), not a user-facing language (future work)

# Language features: Production rules

# Language features: Production rules

GOBLIN grammars have production rules as in CFGs

# Language features: Production rules

GOBLIN grammars have production rules as in CFGs

```
1  <PACKET> ::= <TYPE> <AUX> <PAYLOAD>;
2
3
4
5  <PAYLOAD> ::= <F1> <F2> <BYTES>;
6
7  <BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>;
8
9
10
11
```

# Language features: Type annotations

# Language features: Type annotations

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);
```

# Language features: Refinement types

# Language features: Refinement types

Constrain symbolic terminals with refinement types

```
1  <PACKET> ::= <TYPE> <AUX> <PAYLOAD>;
2
3
4
5  <PAYLOAD> ::= <F1> <F2> <BYTES>;
6
7  <BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>;
8  <TYPE> :: BitVec(8) { <TYPE> = 0x01 or <TYPE> = 0x02; };
9  <BYTE> :: BitVec(8) { <BYTE> bvult 0x88;};
10 <AUX> :: BitVec(8);
11 <F1> :: BitVec(8); <F2> :: BitVec(8); <OPT> :: BitVec(4);
```

# Language features: Semantic constraints

# Language features: Semantic constraints

Attach semantic constraints to production rules

Support types/functions/predicates with SMT-LIB analogues

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

# Language features: Dot notation

# Language features: Dot notation

Reference child nonterminals with dot notation

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

# Language features: Dot notation

# Language features: Dot notation

Dot notation is partial and implicitly universally quantified

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

# Language features: Derived fields

# Language features: Derived fields

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

```
1  <PACKET> ::= <TYPE> <AUX> <PAYLOAD>
2    { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
3  ...
4  <PAYLOAD> ::= <F1> <F2> <BYTES>
5  ...
```

# Language features: Derived fields

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

```
1 <PACKET> ::= <TYPE> <AUX> <PAYLOAD>
2   { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
3 ...
4 <PAYLOAD> ::= <F1> <F2> <BYTES>
5 ...
```

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

# Language features: Derived fields

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

```
1 <PACKET> ::= <TYPE> <AUX> <PAYLOAD>
2   { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
3 ...
4 <PAYLOAD> ::= <F1> <F2> <BYTES>
5 ...
```

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

Cryptographic hashes, checksums, or any computable function

# Language features: Derived fields

```
1 <PACKET> ::= <TYPE> <AUX> <PAYLOAD>
2   { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
3 ...
4 <PAYLOAD> ::= <F1> <F2> <BYTES>
5 ...
```

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

# Language features: Derived fields

```
1 <PACKET> ::= <TYPE> <AUX> <PAYLOAD>
2   { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
3 ...
4 <PAYLOAD> ::= <F1> <F2> <BYTES>
5 ...
```

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

Operators `<-` and `=` are semantically interchangeable

# Language features: Derived fields

```
1  <PACKET> ::= <TYPE> <AUX> <PAYLOAD>
2    { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
3  ...
4  <PAYLOAD> ::= <F1> <F2> <BYTES>
5  ...
```

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

Operators `<-` and `=` are semantically interchangeable

But try telling an SMT solver to find $x$ such that $\text{hash}(x) = y$...

1. Existing approaches to input generation
2. Syntax and language features
3. **Semantics**
4. GOBLIN workflow
   - Well-formedness checks
   - Derived fields
   - Main search algorithm
5. Formal guarantees
6. Evaluation
7. Discussion and future work

# Semantics: Abstract syntax trees

What are the semantics of a GOBLIN grammar?

# Semantics: Abstract syntax trees

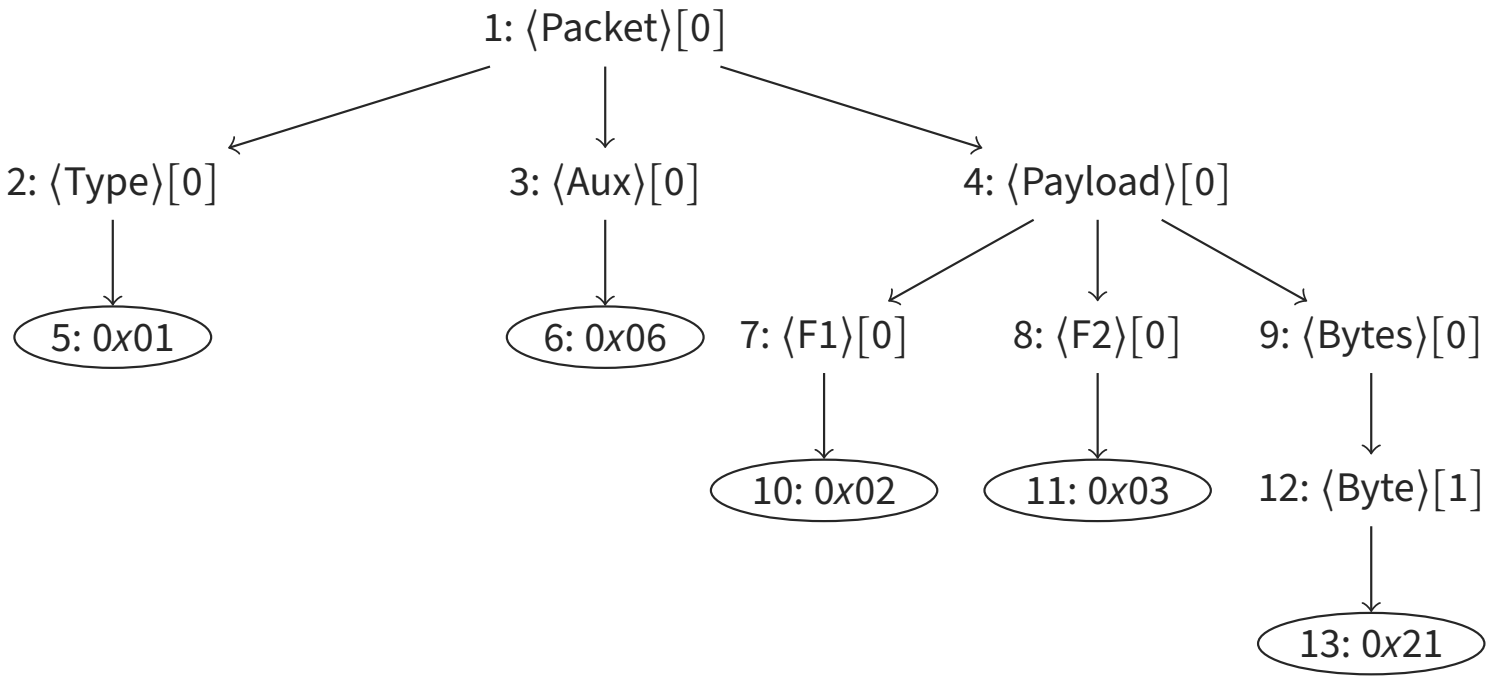
What are the semantics of a GOBLIN grammar?

GOBLIN generates abstract syntax trees, not strings

# Semantics: Abstract syntax trees

What are the semantics of a GOBLIN grammar?

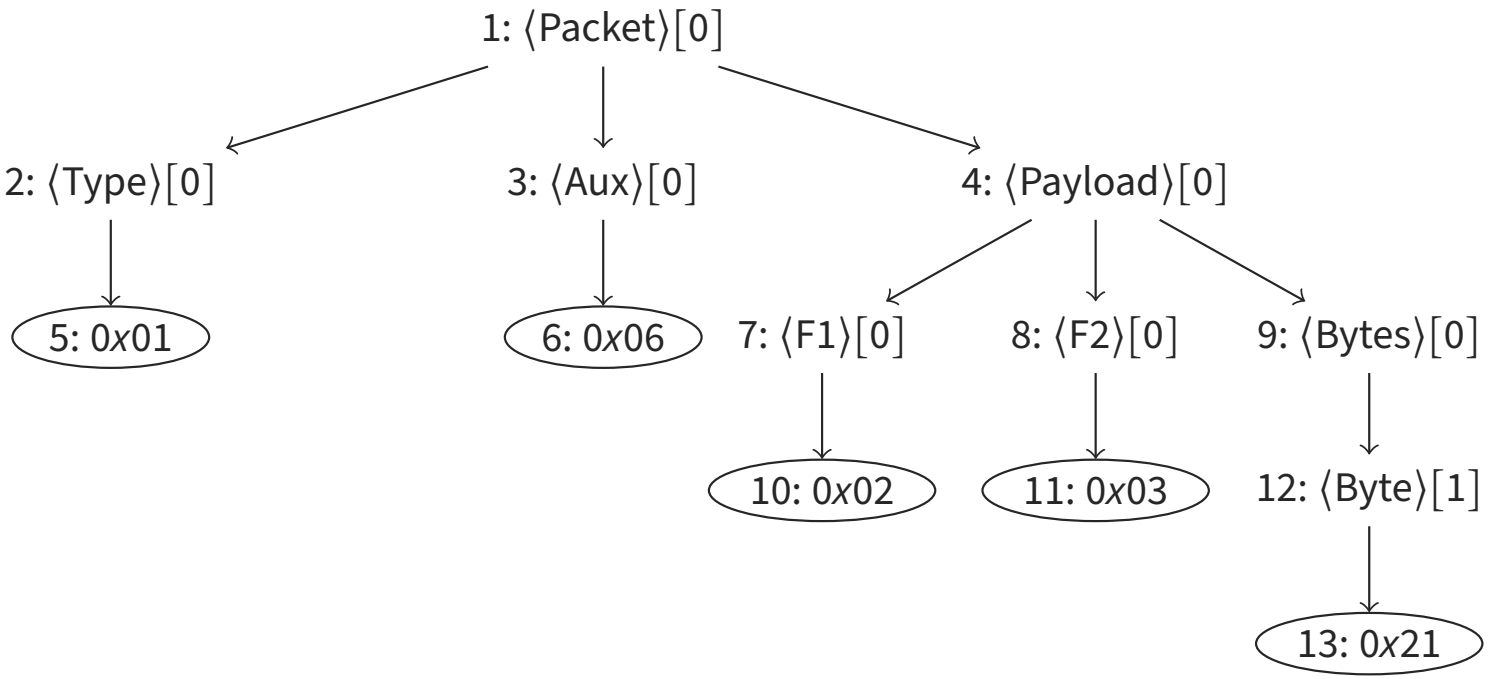
GOBLIN generates abstract syntax trees, not strings



# Semantics: Abstract syntax trees

What are the semantics of a GOBLIN grammar?

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

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

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

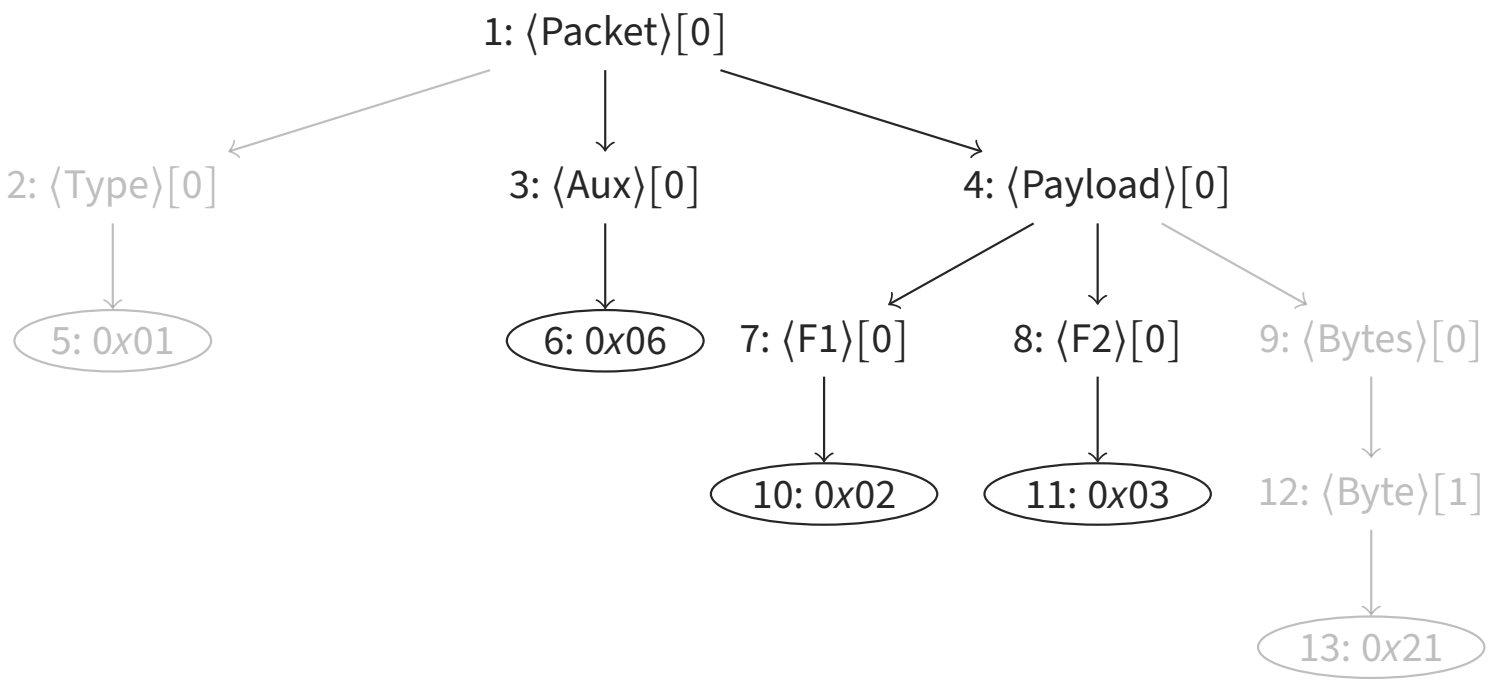
$\mathcal{L}_{\text{AST}}(G) = \{t \mid$

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

$\}$

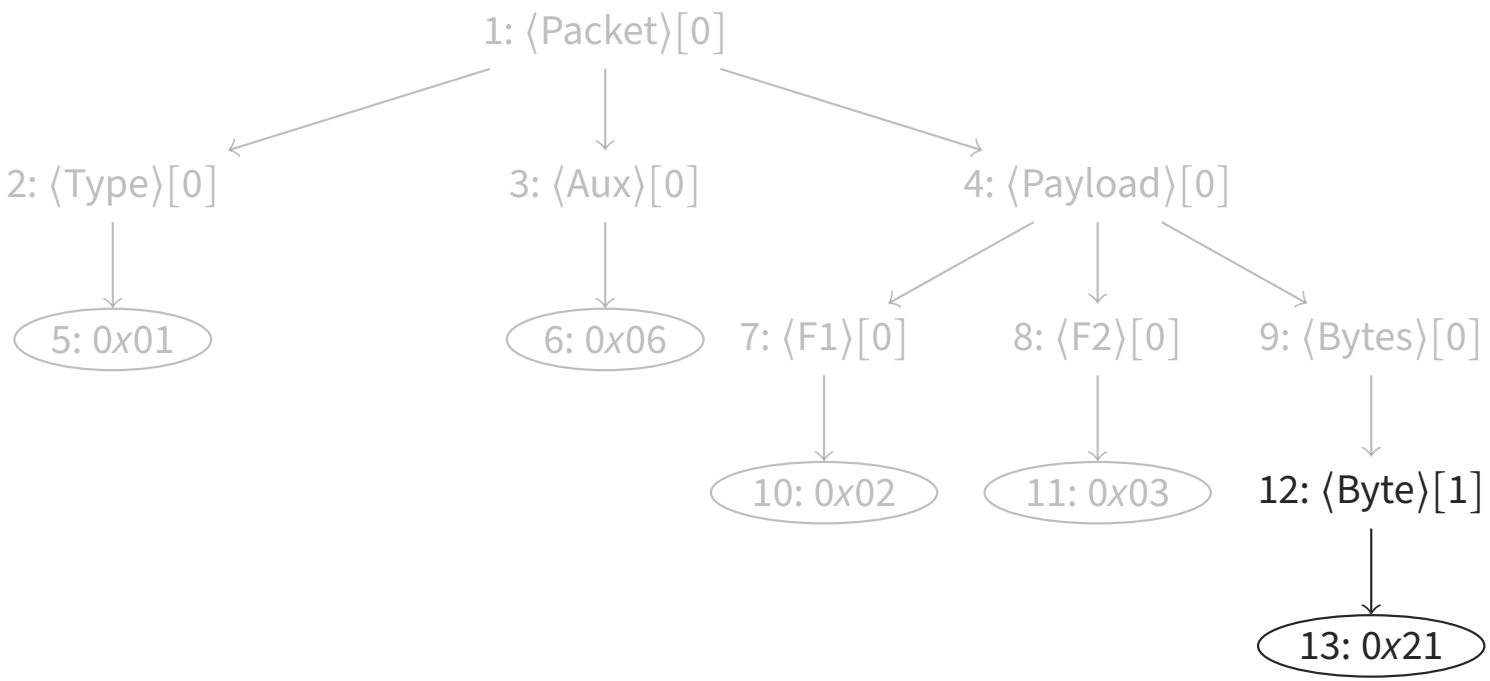
# Semantics: Constraint Satisfaction

```
1  <PACKET> ::= <TYPE> <AUX> <PAYLOAD>
2    { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
3  ...
4  <PAYLOAD> ::= <F1> <F2> <BYTES>
5  ...
```



# Semantics: Constraint Satisfaction

```
1  ...
2  <BYTE> :: BitVec(8) { <BYTE> bvult 0x88; };
3  ...
```



1. Existing approaches to input generation
2. Syntax and language features
3. Semantics
4. **GOBLIN workflow**
   - **Well-formedness checks**
   - **Derived fields**
   - **Main search algorithm**
5. Formal guarantees
6. Evaluation
7. Discussion and future work

# 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 $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

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 $t \in \mathcal{L}_{\mathsf{AST}}(G)$?

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
8. Serialize

# Context-Free Wellformedness

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

# 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

# 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., $S \to S$) 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 $N$

```
1 <SAE_PACKET> ::= <COMMIT> | <CONFIRM>;
2 <COMMIT> ::= <FIELD> <RG_ID_LIST>;
3 <RG_ID_LIST> ::= <RG_ID> <RG_ID_LIST>;
4 <CONFIRM> ::= <FIELD1> <FIELD2>;
5 ...
```

# 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. $\langle A \rangle \leftarrow T[\langle B \rangle], \langle B \rangle \leftarrow T[\langle A \rangle]$)

# Derived Field Checks

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

1. Disallow cyclic dependencies (e.g. $\langle A \rangle \leftarrow T[\langle B \rangle], \langle B \rangle \leftarrow T[\langle A \rangle]$)
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. $\langle A \rangle \leftarrow T[\langle B \rangle], \langle B \rangle \leftarrow T[\langle A \rangle]$)
2. Disallow derived fields in semantic constraints
   - Two stages of computation
   - 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: Main Concepts

Start with context-free case

# Search Algorithm: Main Concepts

Start with context-free case

A derivation tree is an AST that may be open (or closed)

# Search Algorithm: Main Concepts

Start with context-free case

A derivation tree is an AST that may be open (or closed)

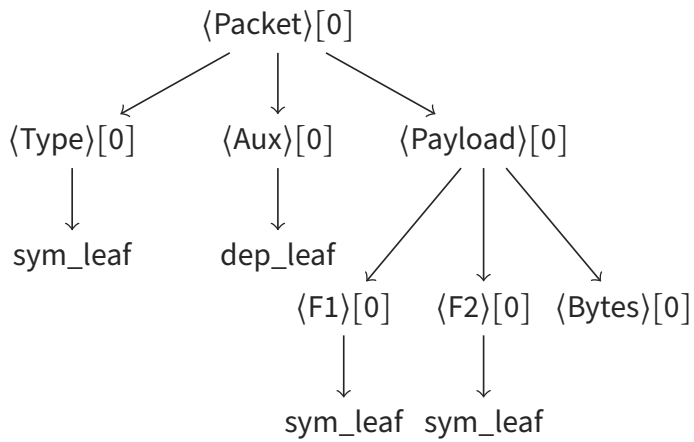
1. Build candidate *dt* via random walk until closed (as in XML)
2. Then instantiate with concrete values

# Search Algorithm: Main Concepts

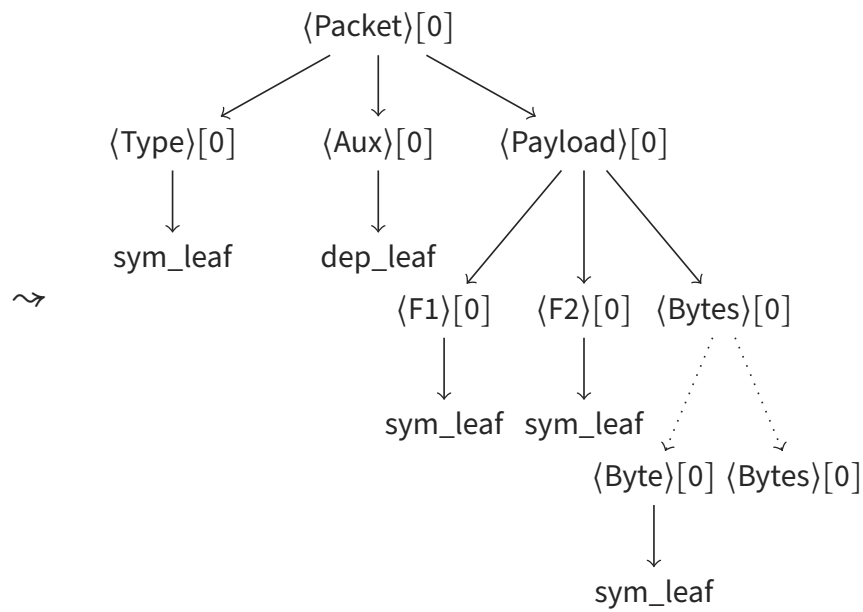
Start with context-free case

A derivation tree is an AST that may be open (or closed)

1. Build candidate *dt* via random walk until closed (as in XML)
2. Then instantiate with concrete values




$dt_1$



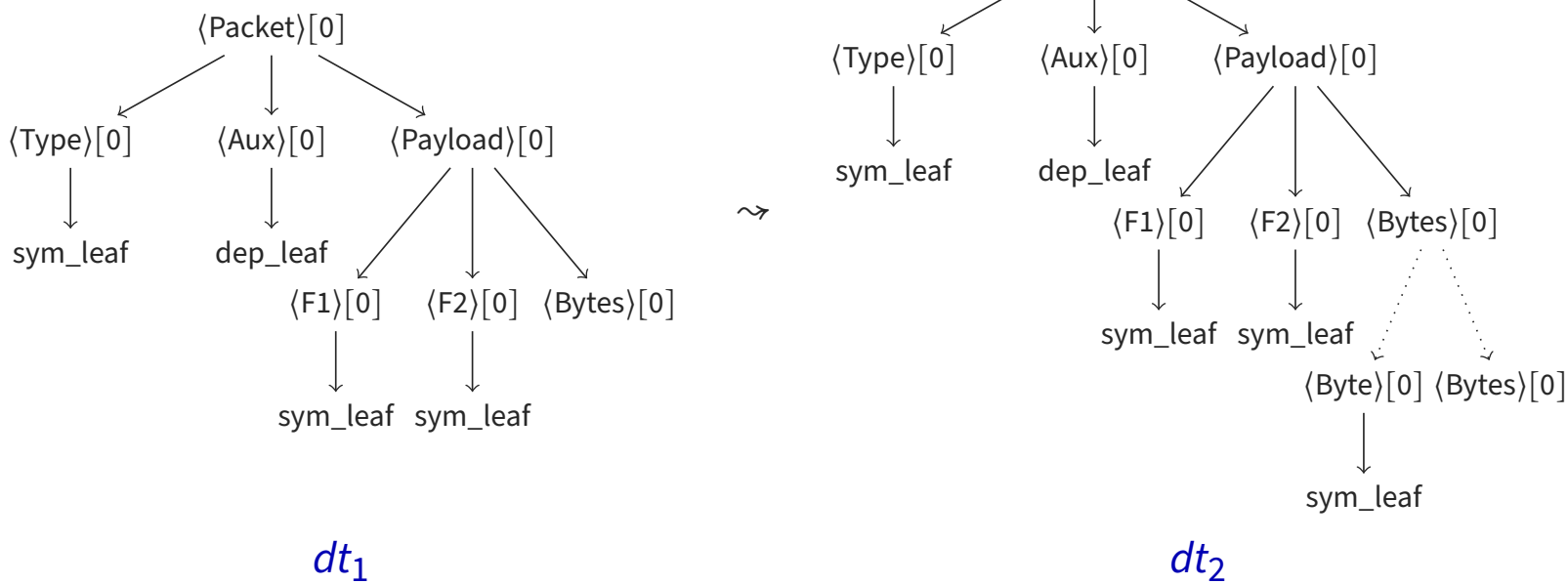

$dt_2$

# Search Algorithm: Main Concepts

Choosing an expansion is called a decision

Decisions are recorded in a decision stack $ds = [dt_1, dt_2]$

Version control helps you undo mistakes!

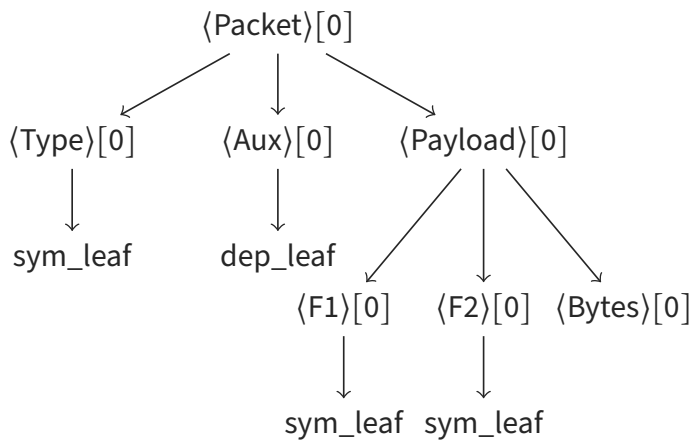



$dt_1$ $dt_2$

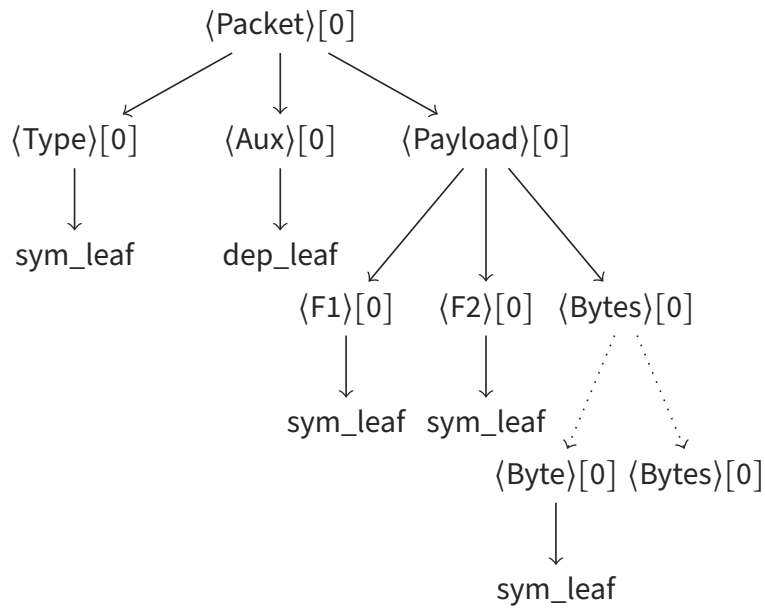
# Search Algorithm: Main Concepts

Some “decisions” are forced (see solid arrows): symbolic terminals and nonterminals with exactly one production rule option

These are called normalization steps and are not stored in *ds*




$dt_1$




$dt_2$

# Search Algorithm: Main Concepts

Termination is not guaranteed...

# Search Algorithm: Main Concepts

Termination is not guaranteed...

**Idea: Bound search depth with IDS**

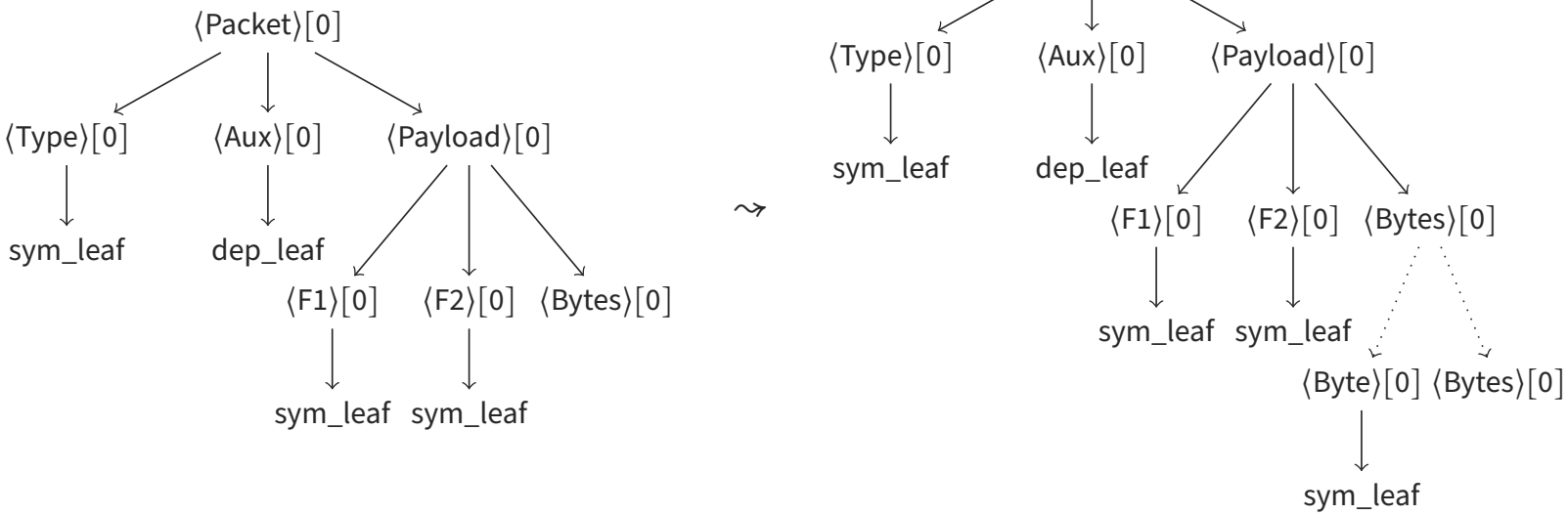
# Search Algorithm: Main Concepts

Pick a depth limit $L$ and associate a search depth with each $dt$

Backtrack once $L$ is exceeded by popping $ds$

Record visited expansions and do not revisit

Restart and increment $L$ if backtracking and $ds = []$



# Search Algorithm: Main Concepts

How to extend the context-free algorithm to handle constraints?

# Search Algorithm: Main Concepts

How to extend the context-free algorithm to handle constraints?

Insight: use an incremental SMT backend

# Search Algorithm: Main Concepts

How to extend the context-free algorithm to handle constraints?

Insight: use an incremental SMT backend

Solver interface

# Search Algorithm: Main Concepts

How to extend the context-free algorithm to handle constraints?

Insight: use an incremental SMT backend

Solver interface

- `(assert c)`: assert a constraint

# Search Algorithm: Main Concepts

How to extend the context-free algorithm to handle constraints?

Insight: use an incremental SMT backend

Solver interface

- `(assert c)`: assert a constraint
- `(push 1)`: push an assertion level

# Search Algorithm: Main Concepts

How to extend the context-free algorithm to handle constraints?

Insight: use an incremental SMT backend

Solver interface

- `(assert c)`: assert a constraint
- `(push 1)`: push an assertion level
- `(pop 1)`: undo assertions in last assertion level (backtrack)

# Search Algorithm: Main Concepts

How to extend the context-free algorithm to handle constraints?

Insight: use an incremental SMT backend

Solver interface

- `(assert c)`: assert a constraint
- `(push 1)`: push an assertion level
- `(pop 1)`: undo assertions in last assertion level (backtrack)
- `(check-sat)`: check if conjunction of active assertions is SAT

# Search Algorithm: Main Concepts

How to extend the context-free algorithm to handle constraints?

Insight: use an incremental SMT backend

Solver interface

- `(assert c)`: assert a constraint
- `(push 1)`: push an assertion level
- `(pop 1)`: undo assertions in last assertion level (backtrack)
- `(check-sat)`: check if conjunction of active assertions is SAT
- `(get-model)`: retrieve a concrete model

# Search Algorithm: Main Concepts

Principles of constraint handling

# Search Algorithm: Main Concepts

Principles of constraint handling

**Idea: utilize incrementality of SMT engine to repair flawed candidate solutions (fear of commitment)**

# Search Algorithm: Main Concepts

Principles of constraint handling

**Idea: utilize incrementality of SMT engine to repair flawed candidate solutions (fear of commitment)**

1. Push an assertion frame at every decision

# Search Algorithm: Main Concepts

Principles of constraint handling

**Idea: utilize incrementality of SMT engine to repair flawed candidate solutions (fear of commitment)**

1. Push an assertion frame at every decision
2. At each decision, `assert` all constraints associated with the expanded *dt* node and call `check-sat`

# Search Algorithm: Main Concepts

Principles of constraint handling

**Idea: utilize incrementality of SMT engine to repair flawed candidate solutions (fear of commitment)**

1. Push an assertion frame at every decision
2. At each decision, `assert` all constraints associated with the expanded *dt* node and call `check-sat`
3. Backtrack and pop an assertion level if `check-sat` yields UNSAT

# Search Algorithm: Main Concepts

Principles of constraint handling

**Idea: utilize incrementality of SMT engine to repair flawed candidate solutions (fear of commitment)**

1. Push an assertion frame at every decision
2. At each decision, `assert` all constraints associated with the expanded *dt* node and call `check-sat`
3. Backtrack and pop an assertion level if `check-sat` yields UNSAT
4. Do not commit to concrete values; call `get-model` once *dt* is closed; then instantiate

# Search Algorithm: Main concepts

But wait, there’s more...

# Search Algorithm: Main concepts

But wait, there's more...

1. Constraints must be disambiguated and universalized

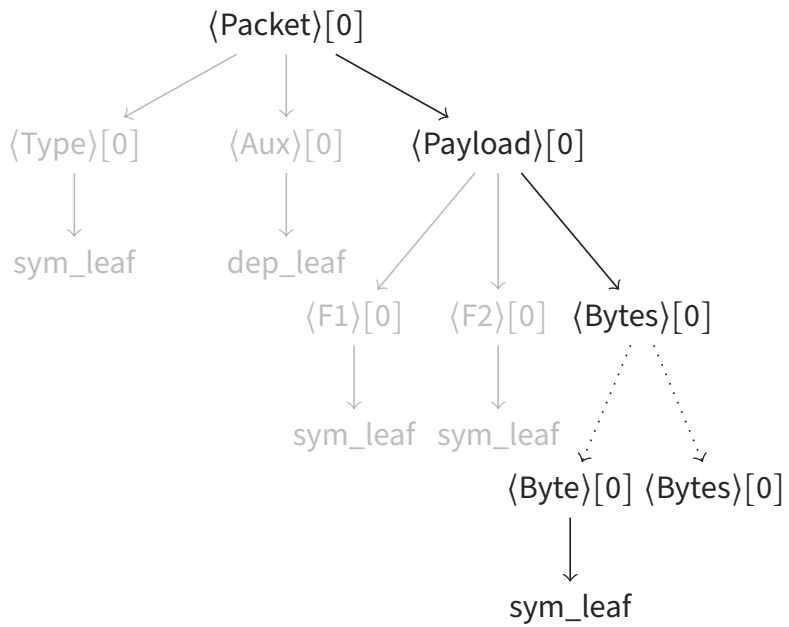
# Search Algorithm: Main concepts

But wait, there's more...

1. Constraints must be disambiguated and universalized

```
1 <PACKET> ::= <TYPE> <AUX> <PAYLOAD> { ... };
2 <PAYLOAD> ::= <F1> <F2> <BYTES>
3  { <BYTES>.<OPT> bvugt 0x0; };
4 <BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>; ...
```

packet0_payload0_bytes0_opt0 $>_{bv}$ 0x0

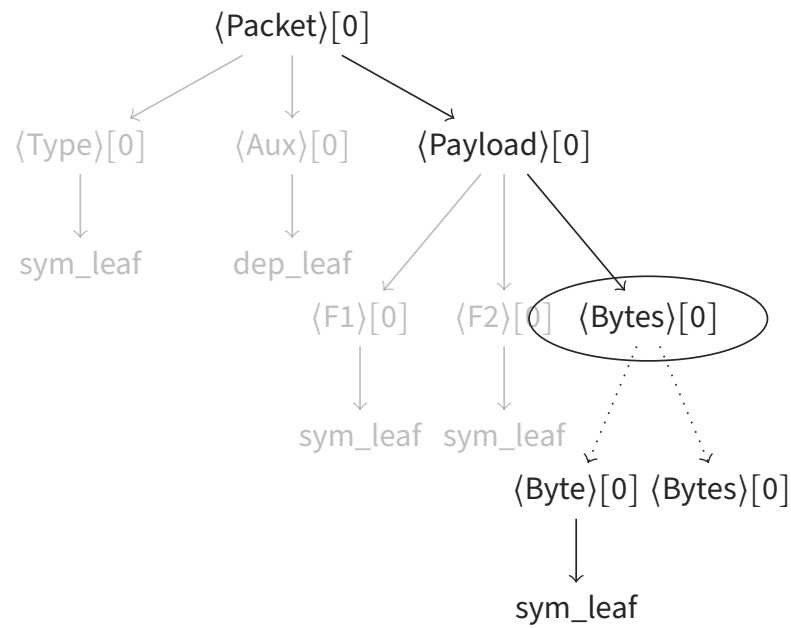


# Search Algorithm: Main concepts

There are wrinkles...

2. This constraint is not applicable!
   - Can it be applicable after a future decision?
   - Extra constraint set $C$ to store inapplicable constraints
   - Remove from $C$ when backtracking

$\texttt{packet0\_payload0\_bytes0\_opt0} >_{bv} 0x0$



1. Existing approaches to input generation
2. Syntax and language features
3. Semantics
4. GOBLIN workflow
   - Well-formedness checks
   - Derived fields
   - Main search algorithm
5. **Formal guarantees**
6. Evaluation
7. Discussion and future work

# 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

1. Existing approaches to input generation
2. Syntax and language features
3. Semantics
4. GOBLIN workflow
   - Well-formedness checks
   - Derived fields
   - Main search algorithm
5. Formal guarantees
6. **Evaluation**
7. Discussion and future work

# 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)

1. Existing approaches to input generation
2. Syntax and language features
3. Semantics
4. GOBLIN workflow
   - Well-formedness checks
   - Derived fields
   - Main search algorithm
5. Formal guarantees
6. Evaluation
7. **Discussion and future**

# 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

# Grammar mutations

How to perform structure-aware mutations on context-sensitive grammars?

- 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 $\mathcal{I}_{tr}(t_i) = \top$

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

$I_{tr}(\langle \text{nt} \rangle[i].\langle \text{nt\_expr} \rangle) = I_{tr'}(\langle \text{nt\_expr} \rangle)$ for $tr'$ rooted at the only $v \in \text{get\_children}(\text{tr}, \text{root}(\text{tr}))$ such that $\ell(v) = \langle \text{nt} \rangle[i]$

...

# 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 $\varphi$; 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

$$[\![G]\!] = \{t \mid t \in \mathcal{L}_{\text{AST}}(G) \land \forall (\text{nt}, \_, \text{constraints}) \in R. \\ \forall s \in \text{get\_subtrees}(t, \text{nt}).\ \forall \varphi \in \text{constraints}.\ s \vDash_{\mathcal{G}} \text{resolve}(\varphi)\}$$

# Calculus

Can conceptualize as guarded rewrite rules on global state

Example rule expands a symbolic terminal

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

# Search algorithm pseudocode

```
1: initializeGlobalState()
2: while ¬allLeavesClosed(dt) do
3:     if there is more than one unvisited expansion then
4:         DECIDE
5:     else
6:         PROPAGATE
7:     while ¬is_normalized(dt) do
8:         NORMALIZEPR (if applicable)
9:         NORMALIZETA (if applicable)
10:    if current search depth d > depth limit L then
11:        if assertionLevel = 1 then
12:            RESTARTDEPTH
13:        else
14:            BACKTRACKDEPTH
15:    else
16:        for all c ∈ constraint set C do
17:            ASSERT if c is applicable
18:        if smt_check_sat() = UNSAT then
19:            if assertionLevel = 1 ∧ ¬bd? then
20:                FAIL
21:            else if assertionLevel = 1 then
22:                RESTARTUNSAT
23:            else
24:                BACKTRACKUNSAT
```