Parse this! Summoning Context-Sensitive Inputs with Goblin

Anonymous Author(s)

Abstract

Grammar-based fuzzers have shown immense promise in identifying bugs in software systems that have highly-structured and intricate input formats (*e.g.*, XML). Many of the existing grammar-based fuzzers rely on context-free grammars (CFGs) to represent the target's input structure. CFGs, however, are often insufficient to precisely capture many application input formats containing context-sensitive constraints. Application-specific fuzzers, albeit effective, lack generality to be adapted to new applications. In this paper, we present Goblin, a new input generation language and tool that helps bridge this gap. Given a context-free grammar annotated with semantic constraints, Goblin generates inputs that both conform to the grammar and satisfy the constraints. While a few prior techniques target this problem, our method is distinguished by: (*i*) support for constraint solving over arbitrary SMT theories (e.g., bitvectors, integers, strings); (*ii*) a minimal core input language with formal semantics that is smaller and less complex than prior work; and (*iii*) a shift from global constraints to local, production rule constraints, which enables easier integration with certain fuzzing workflows. Goblin's input generation approach is inspired by DPLL-style SAT solvers and enjoys the following formal guarantees: *solution soundness*, *solution completeness*, and *refutation soundness*. In addition to comparing Goblin with prior work, we demonstrate its effectiveness by incorporating it into a grammar-based network protocol fuzzer.

1 Introduction

Real-world software systems, such as compilers, file parsers, and network protocol stacks, often have highly complex input specifications. These systems are notoriously difficult to test robustly, and oversights in testing can lead to unintended behavior and security vulnerabilities [23, 25–28]. Fuzzing is a common testing approach, but current approaches face significant limitations. Greybox and whitebox fuzzers leverage information from binaries and source code to help craft test cases. However, these resources may not be available, particularly when testing proprietary software.

In blackbox settings, one then needs to capture the input specification without access to internal information from the system under test. Here, a common approach is **grammar-based fuzzing** [18], where context-free grammars (CFGs) are used to capture input structure. However, while CFGs are often sufficient to describe syntactic requirements on input, they lack the expressiveness to specify **semantic constraints**, which require context sensitivity. For instance, a network message might contain a variable-length payload field f_1 and a length field f_2, where the contents of f_2 must equal the length of f_1 (called a **length constraint**). For many common file formats (e.g. CSV, XML, and more), performing context-free grammar-based fuzzing without taking these semantic constraints into account will lead to virtually none of the generated inputs passing early parsing stages of the system under test [30].

In order to capture more expressive constraints, prior work relies on fuzzers tailor-made for their target applications (e.g., CSmith [33] and Frankencert [8]). While these fuzzers are often very effective, they are not easily generalizable. To that end, we present Goblin (**G**rammar-**o**riented **b**ranching with **l**ogical **i**nference and **n**-threading), a new input generation language and tool that helps bridge this gap. Given a context-free grammar annotated with semantic constraints, Goblin generates inputs that both conform to the grammar and satisfy the constraints. While a few prior techniques target this problem, our method is distinguished by: (*i*) support for constraints over **arbitrary Satisfiability Modulo Theory (SMT) [7] theories** (e.g., bitvectors, integers, strings); (*ii*) a minimal core input language that is smaller and less complex than prior work; and (*iii*) a shift from global constraints to local, production rule constraints, which enables easier integration with certain fuzzing workflows. To our knowledge, Goblin is the first tool to support efficient generation of semantically valid inputs for real-world protocols involving rich constraints over diverse SMT theories in a fully blackbox setting.

Prior Work. The most relevant prior work is ISLa [30], another system for context-sensitive input generation. Much like Goblin, ISLa takes as input both a context-free grammar and a set of context-sensitive constraints and outputs a string in the language of the grammar that also respects the constraints. However, Goblin distinguishes itself from ISLa in several ways. Most notably, Goblin supports constraints over a wider array of data types, has a more minimal input language, considers *local* constraints rather than global constraints, and performs chronological backtracking with an incremental SMT backend (compared to ISLa's nonchronological backtracking with a non-incremental SMT backend).

Fandango [2] also targets semantics-aware input generation, but it uses search-based heuristics in place of constraint solving for higher efficiency. While Fandango is effective for many examples, its effectiveness is highly dependent on the input constraints and built-in set of fitness functions, and it cannot establish formal guarantees.

Our approach is reminiscent of constraint logic programming (CLP), which extends logic programming to also handle constraint satisfaction. In fact, Dewey et al. [14] presented an approach for CLP-based language fuzzing and applied it to JavaScript programs. In principle, Goblin can be framed in terms of CLP—in the full version of the paper [3], we define an alternative semantics of Goblin in terms of a CLP problem. However, we cannot rely on CLP solvers in practice, as they are currently incapable of handling arbitrary SMT theories (e.g., bitvectors, strings) or their combinations.

Approach. We identify three core insights of our approach that distinguish Goblin from previous approaches while maintaining formal guarantees of *solution soundness* (every Goblin output is a member of the language of the input grammar with constraints), *refutation soundness* (if Goblin reports Unsat, then the input grammar's language is empty), and *solution completeness* (if the input grammar's language is nonempty, then Goblin will find a member).

First, our approach conceptually **views context-free grammars as generators of algebraic datatype (ADT) terms rather than**

strings. The ADT view allows each leaf to be assigned a type—e.g., Int, BitVec(n), String—and enables seamless integration with SMT solvers capable of reasoning over these theories. This allows us to specify and natively perform constraint solving in various domains that are inaccessible to ISLa, e.g., bitvector constraints over network packets. In addition, this approach brings the benefits of static typing to the language, while prior work [2, 30] requires *unsafe casting* for non-string constraints.

Second, our approach represents a **minimal core language without sacrificing expressiveness**. Minimizing the set of *core language features* clarifies the formal foundations, simplifies the implementation, and avoids unnecessary special cases. Finally, it provides a solid foundation as an intermediate language for a future, more usable surface-level language. In particular, we envision a future surface-level language which follows in the footsteps of well established and appreciated concepts from theoretical computer science such as attribute grammars and CLP systems.

Third, our approach conceptually views semantic constraints at the **production-rule level** rather than globally. This shift does not affect expressiveness and allows Goblin to integrate naturally with fuzzing workflows involving *grammar-level* mutations, in which grammar rules themselves are mutated rather than concrete inputs. This offers two distinct benefits: (*i*) mutated grammars function as a form of *root cause analysis*; and (*ii*) grammar-level mutations provide an extra mechanism for ensuring diversity of fuzzed inputs.

We build confidence in the correctness of Goblin's main algorithm by framing it as a calculus and proving three theorems: solution soundness, refutation soundness, and solution completeness (for this, see the full version of the paper at [3]).

Evaluation and findings. We perform an experimental analysis with four case studies. Three of them compare directly to prior work (XML, CSV, and Scriptsize C input generation), and they demonstrate Goblin's ability to handle complex constraints and offer competitive performance. The fourth case study (WiFi SAE input generation) demonstrate's Goblin's utility and ability to handle certain constraints and fuzzing workflows that cannot be handled by previous approaches.

Contributions. This paper makes the following contributions: (*i*) we present a general approach for generating inputs that satisfy both context-free syntactic constraints and SMT-expressible semantic constraints; (*ii*) we implement and introduce Goblin, an input generator that supports a minimal yet expressive DSL for defining CFGs with production rule constraints over arbitrary SMT theories; and (*iii*) we evaluate Goblin on four real-world case studies, showing that it scales to complex, constraint-rich input formats.

2 Motivation

We explain how Goblin integrates well with real-world fuzzers, and we discuss the relationship between Goblin and prior work more concretely.

2.1 Integration with Real-World Fuzzers

Grammar-based fuzzers use the implementation-under-test's input language (represented as a grammar) as a guide for generating meaningful test inputs, especially when the test-target has an intricate input format. Without loss of generality, there are two potential classes of grammar-based fuzzers (*i.e.*, Class I and Class II). Class I fuzzers (e.g., [15, 16, 34]) generate a concrete input x from the input language $\mathcal{L}$ such that $x \in \mathcal{L}$. They then apply mutation operations on x to generate new test inputs x's. Class II fuzzers (e.g., [12, 21, 32]) directly mutate the original language $\mathcal{L}$ (*i.e.*, the grammar production rules) to obtain a new language $\mathcal{L}'$. They then aim to obtain concrete test inputs y such that $y \in \mathcal{L}'$. The claimed advantage of Class II fuzzers is that the mutation operations can themselves be input structure aware. In addition, when a concrete input triggers a bug in the system under test, the mutated grammar that was used to generate the input can help with root cause analysis.

Given a language $\mathcal{L}$, generating a concrete input x such that $x \in \mathcal{L}$ is essentially the problem that Goblin solves. The language $\mathcal{L}$ supported by Goblin goes beyond context-free grammars and also includes constraints over the grammar elements. Goblin's Domain-Specific Language (DSL) for capturing the input language $\mathcal{L}$ of the target allows attaching constraints to each production rule of $\mathcal{L}$'s grammar, which we refer to as local, production-level constraints.

Supporting Class II fuzzers in input generators (*e.g.*, Goblin, ISLa, and Fandango) raises a natural question: *If a fuzzer performs grammar production-level mutations, then what should happen to the semantic constraints?* In ISLa [30], the input is a *pair* ⟨ grammar, constraints ⟩, where the constraint language allows quantification, structural predicates over the generated term, and pattern matching over the generated term. In some cases, it may be possible to mutate the constraints in an analogous way to the production rule mutations; however, this is not clear in general. On the other hand, if we attach constraints locally to production rules rather than the entire input grammar, constraint mutation becomes much more natural. Example production rules with local constraints are shown below in Listing 1—the curly braces after each production rule each include a set of constraints that must be satisfied every time the production rule is used in a derivation. More concretely, the first production rule and constraint denote that in any derivation, for every instance of the production rule, the <GROUP_ID> nonterminal must be equal to a bitvector of width 16 with value 13.

Listing 1: Grammar-level mutations

```
<COMMIT> ::= <SEQ> <GROUP_ID> <SCALAR>
  { <GROUP_ID> = int_to_bv(16, 13);
    <SEQ> > int_to_bv(16, 4) };
<RG_CONT> ::= <RG_LENGTH> <RG_TY> <RG_LIST>
  { <RG_LENGTH> <- length(<RG_LIST>); };
 ⤳
<COMMIT> ::= <SEQ> <RG_LIST> <SCALAR>
  { <SEQ> > int_to_bv(16, 4); };
<RG_CONT> ::= <RG_LENGTH> <GROUP_ID>
  { <GROUP_ID> = int_to_bv(16, 13); };
```

Listing 1 also illustrates a *crossover mutation*, where the rules before the arrow denote a fragment of the grammar before mutation, and the rules after the arrow denote the same grammar fragment after mutation. In this mutation, two production rules are selected, and one nonterminal from each production rule is chosen to swap places (here, <GROUP_ID> and <RG_LIST>). It is easy to maintain the semantic constraints over <GROUP_ID> and <SEQ> (they follow the corresponding nonterminals); the lack of clarity is localized to the constraint over <RG_LIST>. In our setting, grammar-level mutations bring more benefits: grammar-level mutations can help boost the diversity of fuzzed inputs, and further, due to the underlying constraint solving, Goblin performs better when performing many shorter generation rounds compared to one long generation round.

2.2 Working Example

We now introduce a working example which we will reference throughout the paper, aiming to give informal intuition for how Goblin works. In Listing 2, we see a grammar and constraints describing the format of a simple network packet.

Listing 2: Working example in Goblin DSL

```
1  <Packet> ::= <Type> <Len> <Payload>
2  { <Len> <- int_to_bv(8, 16 + length(<Payload>));
3    <Type> = int_to_bv(8, 1) =>
4           <Payload>.<Byte> < 32; };
5  <Payload> :: <Byte> <Payload>
6  | <Byte> { (<Byte> bvand 0b11110000)
7                = 0b10100000; };
8  <Type> :: BitVec(8)
9  { <Type> = int_to_bv(8, 0) lor
10   <Type> = int_to_bv(8, 1); };
11 <Len> :: BitVec(8);
12 <Byte> :: BitVec(8);
```

Line 1 describes a production rule for <Packet>, a single option producing three nonterminals <Type>, <Len>, and <Payload>. Lines 2, 3, and 4 (surrounded by curly braces) denote semantic constraints on the production rule—that is, a **length constraint** which expresses that <Len> should have a value equal to sixteen plus the length of the <Payload>, and another constraint that restricts the length of the <Payload> when <Type> takes value 1. The semicolon separating the constraints should be interpreted conjunctively. Lines 8, 11, and 12 denote **type annotations**, which ascribe types to leaf symbols in the grammar. As in lines 9 and 10, type annotations can also carry semantic constraints (which we call **refinement types**).

2.3 Comparison with Prior Work

Goblin and ISLa. As mentioned in Section 1, the most relevant prior work is ISLa [30], another system for context-sensitive input generation. Goblin distinguishes itself from ISLa in four main ways.

First, the semantics of ISLa restrict it to only natively supporting string and integer constraints. This means that the example presented in Listing 2 could not be directly expressed in ISLa. Technically, one could encode the bitwise and constraint (bvand) in terms of string constraints at the SMT-LIB level, but this would be incredibly tedious and less performant. Further, the support for integer constraints is limited in the sense that it relies on unsafe string to integer casts. In fact, in ISLa's formal semantics [30], the semantics of quantification over integers is defined as quantification over strings that can be casted to integers. ISLa's reliance on strings comes from the fact that ISLa conceptually views grammars as generating *strings* rather than *ADT terms*, and so each nonterminal symbol has a string type. In addition to reliance on unsafe casting, this introduces an obstacle for SMT constraints over string variables representing string nonterminal symbols—the SMT solver may return a model where the string variable does not respect the input CFG. To deal with this, ISLa approximates the input CFG as a regular expression, a process they call "grammar to regex" [30]. With the ADT view, Goblin sidesteps this issue entirely. While ISLa's string and integer constraints are sufficient for some domains (e.g., compiler fuzzing), it is inadequate for applications with non-string, non-integer constraints (e.g., network protocol stacks).

Second, the Goblin language is minimal compared to ISLa's. Many features that are part of ISLa's core language are absent from Goblin, but can still be encoded. For example, ISLa considers **structural constraints** which constrain the structure of the generated term. More concretely, ISLa can encode **definition before use** constraints for compiler input generation through a predicate `before` and existential quantification, both native to the input language. However, Goblin can express these constraints, despite not having built-in structural predicates or existential quantification. Because Goblin is an *ADT term generator* with *local constraints*, we can encode such constraints akin to how one would do so in a CLP system or attribute grammar. Minimality is not necessarily a benefit or end goal in itself—however, we argue that a well-designed *intermediate language* serves to to clarify the formal foundations, simplify the implementation, and avoid unnecessary special cases. In the future, we envision designing and supporting a surface-level input language with more usability features and syntactic sugar.

Third, Goblin supports local production rule constraints rather can global constraints, as discussed in Section 2.1.

Fourth, Goblin's style of backtracking is chronological, while ISLa's is nonchronological. With chronological backtracking, Goblin incrementally expands and backtracks a candidate solution, which enables an incremental SMT backend. This backend enables Goblin to both detect unsatisfiable states quickly and update partial solutions as new constraints are encountered.

Goblin and Fandango. Fandango [2] also targets semantics-aware input generation, but it uses search-based heuristics in place of constraint solving for higher efficiency. Fandango is incredibly efficient for many examples, and its constraint language (general Python code) has unparallel expressivity. However, Fandango's effectiveness is highly dependent on the input constraints and cannot establish formal guarantees. Conceptually, the approach produces instances using a genetic algorithm which attempts to steer towards semantically correct instances through fitness functions that favor those instances that are the "closest" to satisfying the semantic constraints. However, synthesizing suitable fitness functions from scratch based on the input grammar and constraints is currently out of reach, so Fandango [2] chooses from a set of built-in fitness functions. If none of the built-in fitness functions works for an input, Fandango struggles to produce even a single valid instance. Several simple examples demonstrate this phenomenon—for example, (*i*) an input grammar with two nonterminals <nt1> and <nt2> with a single semantic constraint `<nt1> == <nt1>` (one equality constraint), (*ii*) an input grammar with two nonterminals with a single semantic constraint `str(<nt1>) == str(<nt1>) + "foo"` (one string constraint), and (*iii*) an input grammar with an integer nonterminal and a single semantic constraint `int(<nt>) in {1, 2, 3}` (one set membership constraint). In addition, when we tried to use Fandango to synthesize a valid input for the WiFi SAE packet grammar and constraints that we explore as part of our experimental evaluation of Goblin (see Section 8), Fandango was not able to produce a single instance. The Fandango examples demonstrating our encodings for these problems is available at [3].

In short, choosing or synthesizing a suitable fitness function for any arbitrary set of input constraints is an extremely difficult problem. It could potentially be remedied through user-supplied fitness functions, but this detracts from the generality of the approach. If the constraints are non-monotonic, a suitable fitness function may not exist in the first place.

3 Design Overview

Next, we give an informal overview of Goblin's input generation process. The main pipeline is depicted in Figure 1, and formal description follows in the full version of the paper [3].

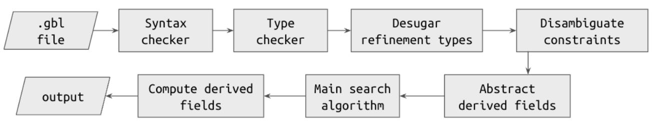

Figure 1: Main pipeline.

Context-free language emptiness. As a preprocessing step, with the Syntax checker module, we implement a standard algorithm to determine whether or not the *context-free* version of the input grammar (i.e., without any extra semantic constraints) has an empty language [19]. At the context-free level, language emptiness stems from non-well-founded recursion (i.e., recursion without a base case), leading to infinite derivations. Intuitively, the algorithm proceeds by iteratively expanding a set of nonterminals that are known to be able to produce terminal strings until reaching a fixpoint, and then checking if the set contains the start symbol. In fact, we perform a more conservative check by mandating that *all* reachable nonterminals must be a member of this set, as violation likely signals a modeling error.

Listing 3: Syntax checks

```
<SAE_PACKET> ::= <COMMIT> | <CONFIRM>;
<COMMIT> ::= <FIELD1> <RG_ID_LIST>;
<RG_ID_LIST> ::= <RG_ID> <RG_ID_LIST>;
<CONFIRM> ::= <FIELD2> <FIELD3> <FIELD4>;
...
```

An example input rejected by our syntax checker (inspired by real experience with our WiFi case study) is in Listing 11. While the language of the grammar is non-empty through a derivation involving <CONFIRM>, the <COMMIT> nonterminal cannot derive a finite string due to a missing base case in the definition of <RG_ID_LIST>. Hence, the input is conservatively rejected, as no finite derivation through <COMMIT> is possible.

Other syntax checks. Other syntax checks include the detection of dangling identifiers and invalid dot notation references.

Type checking. Because the language requires type annotations for leaf-level nonterminals, we can perform standard bidirectional type checking techniques with two mutually recursive functions check_type_expr and infer_type_expr.

Listing 4: Type checking

```
1 <PACKET> ::= <F1> <F2> <ALGO>
2   { <F1> = <F2> bvplus <ALGO>.<ALGO_ID>; };
3 <ALGO> ::= <ALGO_ID> <ALGO_SETTINGS> ;
4 <F1> :: BitVec(16); <F2> :: BitVec(16);
5 <FIELD3> :: BitVec(16); <ALGO_ID> :: BitVec(16);
6 <ALGO_SETTINGS> :: BitVec(8);
```

For example, in Listing 4, to perform type checking of the constraint on line 2, we call infer_type_expr to infer the type of the whole expression, and we call check_type_expr to see if it matches the expected type, which is always Bool. Type inference is necessary in our setting since the bitvector types are *dependent types* which depend on the length of the bitvector. Notice that we cannot tell if the subexpression with the bvplus operation is well-typed without inferring the types of both arguments and checking if their lengths are equal. With this type checking, we can specify constraints over network packets without having to perform unsafe casts from strings to bitvector types, where the input string may or may not be convertible to a bitvector, and bitvector operations may not be well-defined for the given operands depending on their lengths.

Refinement type constraints. Goblin's input language supports semantic constraints on type annotations, which we call *refinement types*. We support refinement types by desugaring type annotation constraints to production rule constraints by "inlining" the type annotation constraint for some nonterminal <nt> at every production rule option containing <nt> (on the right-hand side).

Listing 5: Refinement types

```
<S> ::= <NAT> <NAT>;
<NAT> :: Int { <NAT> >= 0; };
 ⇝
<S> ::= <NAT> <NAT> { <NAT> >= 0; };
<NAT> :: Int;
```

We demonstrate the inlining with a simple example in Listing 5.

Disambiguating dot notation. Notice that in Listing 5, after inlining, the constraint <NAT> >= 0 applies to a production rule with two instances of the <NAT> nonterminal. Under the hood, the constraint is disambiguated to <NAT>[0] >= 0 and <NAT>[1] >= 0, where we take the semantics of *implicit universal quantification*. This is discussed in more detail in Section 4.

Derived fields. Some constraints are not amenable to analysis by SMT solvers. For example, checksums, constraints over cryptographic fields, and constraints involving non-linear arithmetic are highly expensive and/or difficult to translate to SMT-LIB [6]. We handle these constraints separately from the rest of the search problem through a language feature called *derived fields*, in which syntax <field> <- <expression> denotes that the value for nonterminal <field> can be computed with expression <expression>. For example, in Listing 2, the <Len> nonterminal is a derived field since its value can be calculated as defined on line 2. To make derived fields computable outside the solver, we need to detect and reject two scenarios: (*i*) we disallow mutual (cyclic) dependencies between derived fields, and (*ii*) we disallow including derived fields in (non-derived) semantic constraints. To detect cyclic dependencies, we borrow an efficient technique of circularity detection in attribute grammars [10, 35]. We build a directed graph for each production rule. There is a node for each nonterminal on the right-hand side of the production rule and an edge from ⟨`nt1`⟩ to ⟨`nt2`⟩ when ⟨`nt1`⟩ is a derived field defined in terms of ⟨`nt2`⟩. If this graph contains a cycle, then the input is rejected for a cyclic dependency. This algorithm is a fast over-approximation that is computable in polynomial time, but works for "every practical example" [10]. To detect when derived fields are mentioned in (non-derived) semantic constraints, we simply scan every semantic constraint for dot notation expressions mentioning nonterminals that are associated with derived fields.

Listing 6: Derived fields

```
<S> ::= <LEN> <TYPE> <PAYLOAD>
{ <LEN> <- length(<PAYLOAD);
  <LEN> < 64; };
<LEN> :: BitVec(8); <TYPE> :: BitVec(8);
<PAYLOAD> :: List(Bool);
```

The example in Listing 6 is rejected since derived field `<LEN>` is mentioned in the semantic constraint `<LEN> < 64`. In this case, the user can remedy the situation by replacing the derived field operator `<-` with an equality constraint. (Also, in Listing 6, if `<TYPE>` were a derived field defined by an expression containing `<LEN>`, then there would be a cyclic dependency between `<TYPE>` and `<LEN>`.) After these checks, these derived fields are removed from the grammar and replaced with opaque, "stub" nonterminals that record the position of the derived field with symbolic leaf symbols rather than the input production rules.

Listing 7: Derived fields

```
<S> ::= sym_leaf <TYPE> <PAYLOAD>;
<TYPE> :: BitVec(8); <PAYLOAD> :: List(Bool);
```

Assuming `<LEN>` remains a derived field, Listing 6 gets translated to Listing 7, where the `<LEN>` nonterminal is replaced by a special symbolic leaf value. The `sym_leaf` value will be present in the term generated by the main search algorithm, say, `(S (LEN sym_leaf) (TYPE 0b00000000) (PAYLOAD []))`. Here, we can compute the value of `sym_leaf` to give output term `(S (LEN 0b00000000) (TYPE 0b00000000) (PAYLOAD []))`.

Polymorphic length function. Derived fields enable language features that don't rely on SMT support. An example is Goblin's *polymorphic length function*, which computes the length of an input nonterminal of any type. Supporting such an operator at the SMT-LIB level is difficult—one strategy is to synthesize an ADT for the input nonterminal and then synthesize a length function that takes a term of that ADT as input. However, we can support this easily in Goblin by simply including evaluation rules in Goblin's internal expression evaluator (used for computing derived fields).

Listing 8: Polymorphic length

```
<S> ::= <LEN> <PAYLOAD>
 { <LEN> <- length(<PAYLOAD>); };
<PAYLOAD> :: <F1> <F2> <L>;
<F1> :: BitVec(16); <F2> :: BitVec(16);
<L> :: List(Bool);
```

Listing 8 demonstrates the application of the polymorphic `length` function to a nonterminal containing two bitvectors and a list of Booleans. The `length` function works by adding all the lengths of the concrete leaf values of the input term (string length for strings, bit width for bitvectors, and so on). Here, it would compute a value equal to the length of `<L>` plus 32.

Main solving algorithm. The main search algorithm will be described in detail in Section 5. At a high level, the solving algorithm iteratively builds a *candidate solution* by walking through the grammar and choosing production rules to expand. Whenever applicable semantic constraints are encountered, they are eagerly asserted to an SMT solver to detect unsatisfiable states as quickly as possible. When unsatisfiable states are detected, the search backtracks to the most recent decision (i.e., most recent choice of production rule to expand) if possible. Otherwise, the search will either report that the input problem is unsatisfiable, or continue the search, depending on greater context.

Serialization. Goblin produces ADT terms rather than strings. More specifically, Goblin produces Lisp-style terms, where each term is either (*i*) a concrete leaf-level value (say, `1000` or `"foo"`), (*ii*) a symbolic leaf-level value for derived fields (say, `sym_leaf_27`), or (*iii*) a constructor string paired with a list of subterms (say, `(S (LEN 0b00000000) (TYPE 0b00000000) (PAYLOAD [])))` from the **Derived fields** paragraph. To produce strings, the user defines a serialization function mapping these Lisp-style terms to strings outside of Goblin.

4 Goblin's Language and Semantics

In this section, we give a more precise characterization of Goblin's syntax and semantics.

4.1 Syntax

We define the language syntax using a standard extension of Backas-Naur form [4] where the * operator denotes zero or more instances, the + operator denotes one or more instances, and the square brackets denote optional elements.

```
          ⟨S⟩ ::= ⟨element⟩+
    ⟨element⟩ ::= ⟨ty_annot⟩ | ⟨prod_rule⟩
   ⟨ty_annot⟩ ::= ⟨nt⟩ :: ⟨type⟩ [ { ⟨constraint⟩+ } ]
  ⟨prod_rule⟩ ::= ⟨nt⟩ ::= ⟨nt⟩+ [ { ⟨constraint⟩+ } ]
                  [ | ⟨nt⟩+ [ { ⟨constraint⟩+ } ] ]+;
 ⟨constraint⟩ ::= ⟨derived_field⟩; | ⟨expr⟩;
⟨derived_field⟩ ::= ⟨nt⟩ <- ⟨expr⟩
       ⟨expr⟩ ::= ⟨expr⟩ ⟨binop⟩ ⟨expr⟩ | ⟨unop⟩⟨expr⟩ |
                  ⟨f⟩(⟨expr⟩, ..., ⟨expr⟩) | ⟨p⟩(⟨expr⟩, ..., ⟨expr⟩) |
                  ⟨nt_expr⟩ | constant
    ⟨nt_expr⟩ ::= ⟨nt⟩ | ⟨nt⟩.⟨nt_expr⟩
         ⟨nt⟩ ::= < identifier >
       ⟨type⟩ ::= Bool | Int | String | Set(⟨type⟩) ...
          ⟨f⟩ ::= length | bv_cast | ...
          ⟨p⟩ ::= leq | geq | ...
      ⟨binop⟩ ::= land | lor | ...
       ⟨unop⟩ ::= lneg | minus | ...
```

In the BNF, we did not enumerate all the types, predicate symbols, function symbols, and constants. In principle, we support every type, predicate symbol, and function symbol that has a direct translation to SMT-LIB [6] or cvc5's [5] support for non-standard theories [11]. (In practice, we have not yet implemented support for all of the types, function symbols, and predicate symbols, but will glady extend the input language where demand arises.)

Dot notation. Before formally defining the language semantics, we first informally discuss a tricky aspect of the language: **ambiguous dot notation references**. Consider Listing 9.

Listing 9: Ambiguous dot notation reference

```
<A> -> <B> <B> <C> { <B>.<D> > <C>; };
<B> -> <D> <D>;
<D> :: Int;
```

Above, the **nonterminal expression** <B>.<D> intuitively could refer to either the first or second child <D> of either the first or second occurrence of <B>. Goblin treats all ambiguous references of this form as **implicitly universally quantified** over the structure of generated terms—that is, one can view the above constraint as internally desugaring to `<B>[0].<D>[0] > <C>[0]; <B>[0].<D>[1] > <C>[0]; <B>[1].<D>[0] > <C>[0]; <B>[1].<D>[1] > <C>[0]`, where the bracket notation $[i]$ of a nonterminal symbol uniquely indicates which occurrence of the nonterminal symbol is being referenced. Furthermore, also consider Listing 10, where <B> gets a separate production rule also referencing <D>.

Listing 10: Ambiguous dot notation reference II

```
<A> -> <B> <B> <C> { <B>.<D> > <C>; };
<B> -> <D> <D> | <D>;
<D> :: Int;
```

At term generation time, if <B>'s second production rule is chosen, then (e.g.) constraint `<B>[0].<D>[1] > <C>[0]` is considered trivially satisfied, since `<B>[0]` does not have a child `<D>[1]` (instead, it has a single child, `<D>[2]`).

4.2 Semantics

To capture the language semantics, we first formally define an **augmented context-free grammar** (ACFG) G. Intuitively, G is a context-free grammar (CFG) where each production rule optionally carries additional (context-sensitive) constraints, and where we replace nonterminal symbols with type annotations, which specify that the given nonterminal symbol ranges over the given type.

More concretely, G is a 4-tuple (N, R, Γ, S), where N is a finite set of nonterminal symbols, $R : N \times N^* \times C^*$ is a relation capturing the production rules where each production rule additionally carries zero or more semantic constraints, Γ is a set of $N \times \Lambda$ pairs denoting type annotations, and $S \in N$ is the distinguished start symbol. Additionally, we require that each nonterminal symbol either (mutually exclusively) is mentioned on the left-hand side of one or more production rules, or has exactly one type annotation.

Rather than working with raw strings, we define the semantics of an ACFG G as the set of valid abstract syntax trees producible by the grammar (denoted $\mathcal{L}_{\text{AST}}(G)$). We define an **abstract syntax tree** (AST) tr for a given ACFG G as a 3-tuple $(V, \vec{E}, \ell)$, where $(V, \vec{E})$ denotes a directed graph without (even undirected) cycles, and $\ell :: V \to N \cup (\bigcup \Lambda)$ denotes the label of a given vertex with its associated nonterminal symbol, or value ranging over its given type in G. We abuse notation to allow ℓ to also return a nonterminal symbols associated integer index for disambiguation (as discussed in Section 4.1) where convenient. Also, get_children $:: AST \times V \to V^*$ takes the expected semantics, and we abuse notation by using $\in$ to mean both set and list membership. Additionally, tr has two syntactic well-formedness requirements:

- (i) $\ell(\text{root}(G)) = S$ (the AST is rooted at the start symbol);
- (ii) for every $v \in V$, either
  - (a) for every $v_1, \ldots, v_n \in \text{get\_children}(tr, v)$, $(\ell(v), (\ell(v_1), \ldots, \ell(v_n)), \_) \in R$, or
  - (b) there is only a single $v_1 \in \text{get\_children}(tr, v)$ and $(\ell(v), \tau) \in \Gamma$, or
  - (c) v is a leaf vertex with $\ell(v) \in \tau$, where τ is the type of $\ell(\text{parent}(v))$ ($\ell(\text{parent}(v))$ must be associated with a type annotation).

Informally, each node in the tree is either a non-leaf node with children representing an application of some production rule in G, a non-leaf node with a single child representing some type annotation in G, or a well-typed leaf.

Next, we define the meanings of terms in a production rule constraint for a given AST tr by defining interpretation function $\mathcal{I}_{tr}(t)$, which outputs the denotation of term t in AST tr. It is analogous to first-order logic, except (i) the variable assignment is determined by the structure and labels of tr rather than by a mapping of variable names to values, (ii) terms evaluate to a distinguished symbol $\top$ if they reference nonterminal expressions that are not present in the given tree, and (iii) we assume all terms are well-typed (e.g., primitive values take the expected types, each function symbol has a given set of input types and an output type, and each function symbol is passed the correct number of arguments with the correct types). For primitive values (e.g., integers), the denotation is defined by the identity function. Otherwise,

$$
\begin{aligned}
&\mathcal{I}_{tr}(\langle \mathsf{nt} \rangle[i]) = \top \text{ if}\\
&\quad nt, \_, \_ \in R \text{ and}\\
&\quad \{v \in \mathsf{get\_children}(tr, \mathsf{get\_root}(tr)) \mid \ell(v) = \langle \mathsf{nt} \rangle[i]\} = \emptyset\\
&\mathcal{I}_{tr}(\langle \mathsf{nt} \rangle[i]) = \ell(v) \text{ for the unique}\\
&\quad v \in \mathsf{get\_children}(tr, \mathsf{get\_root}(tr)) \text{ if } nt, \_ \in \Gamma\\
&\mathcal{I}_{tr}(\langle \mathsf{nt} \rangle[i]) = \mathcal{I}_{tr'}(\langle \mathsf{nt} \rangle[i]) \text{ for } tr' \text{ rooted at the unique}\\
&\quad v \in \mathsf{get\_children}(tr, \mathsf{get\_root}(tr)) \text{ such that } \ell(v) = \langle \mathsf{nt} \rangle[i]\\
&\quad \text{if } nt, \_, \_ \in R\\
&\mathcal{I}_{tr}(\langle \mathsf{nt} \rangle[i].\langle \mathsf{nt\_expr} \rangle) = \top \text{ if}\\
&\quad \{v \in \mathsf{get\_children}(tr, \mathsf{get\_root}(tr)) \mid \ell(v) = \langle \mathsf{nt} \rangle[i]\} = \emptyset\\
&\mathcal{I}_{tr}(\langle \mathsf{nt} \rangle[i].\langle \mathsf{nt\_expr} \rangle) = \mathcal{I}_{tr'}(\langle \mathsf{nt\_expr} \rangle), \text{ for } tr' \text{ rooted at the unique}\\
&\quad v \in \mathsf{get\_children}(tr, \mathsf{get\_root}(tr)) \text{ such that } \ell(v) = \langle \mathsf{nt} \rangle[i]\\
&\mathcal{I}_{tr}(f(t_1, \ldots, t_n)) = \top \text{ if some } \mathcal{I}_{tr}(t_i) = \top\text{; otherwise,}\\
&\mathcal{I}_{tr}(f(t_1, \ldots, t_n)) = \mathcal{I}_{tr}(f)(\mathcal{I}_{tr}(t_1), \ldots, \mathcal{I}_{tr}(t_n))
\end{aligned}
$$

We define a satisfaction relation $\models_{\mathcal{G}}$ that captures whether or not a given constraint in G is satisfied by a given abstract syntax tree. It proceeds as in quantifier-free first-order logic, except it also uses $\top$ analogously to the term semantics.

$$
\begin{aligned}
&\mathcal{I}_{tr} \models_{\mathcal{G}} \varphi \text{ if some subterm } t_i = \top\text{; otherwise,}\\
&\mathcal{I}_{tr} \models_{\mathcal{G}} p(t_1, \ldots, t_n) \text{ if } (\mathcal{I}_{tr}(t_1), \ldots, \mathcal{I}_{tr}(t_n)) \in \mathcal{I}_{tr}(p)\\
&\mathcal{I}_{tr} \models_{\mathcal{G}} \neg\varphi \text{ if } \mathcal{I}_{tr} \not\models_{\mathcal{G}} \varphi\\
&\mathcal{I}_{tr} \models_{\mathcal{G}} \varphi_1 \wedge \varphi_2 \text{ if } \mathcal{I}_{tr} \models_{\mathcal{G}} \varphi_1 \text{ and } \mathcal{I}_{tr} \models \varphi_2
\end{aligned}
$$

While we presented the semantics for predicate symbols and function symbols generically, the Goblin's actual predicate and function symbols have fixed interpretations and take the expected semantics.

Below, $\mathsf{CFG}(\cdot)$ is a function that takes an ACFG as input and removes its semantic constraints, returning a standard CFG. Also, $\mathsf{get\_subtrees}(\cdot, \cdot)$ is a function that takes as input an AST t and a nonterminal symbol $\langle \mathsf{nt} \rangle$, and returns all subtrees in t rooted at vertices v with $\ell(v) = \langle \mathsf{nt} \rangle$. Finally, resolve $:: C \to \mathcal{P}(C)$ performs the desugaring of ambiguous dot notation references discussed previously, and we lift $\models_{\mathcal{G}}$ to sets of constraints as expected. Put together, the semantics of an input, $[\![G]\!]$, is defined as the set of ASTs in the language of G that also satisfy all the semantic constraints.

$$
\begin{aligned}
[\![G]\!] = \{t \mid\ & t \in \mathcal{L}_{\mathsf{AST}}(\mathsf{CFG}(G)) \wedge \forall(\mathsf{lhs}, \_, \mathsf{constraints}) \in R.\\
&\forall s \in \mathsf{get\_subtrees}(t, \mathsf{lhs}).\ \forall \varphi \in \mathsf{constraints}.\\
&s \models_{\mathcal{G}} \mathsf{resolve}(\varphi)\}
\end{aligned}
$$

5 Goblin's Constraint Solving Approach

Here, we describe Goblin's core algorithm for constraint solving, continuing to reference our working example from Listing 2.

Basic search strategy. Intuitively, our search strategy mimics a DPLL-style [13] search in which a candidate solution, which we call a **derivation tree** (formally defined in the full version of the paper [3]), is incrementally constructed through node expansions and backtracking until either a solution is found or the input grammar G is deemed unsatisfiable (i.e., $[\![G]\!] = \emptyset$). For conciseness, we represent derivation trees as strings denoting the expansion so far, where open leaves in the derivation tree are represented by nonterminal symbols. We can consider an open derivation of the working example in Listing 2 as <Type> <Len> sym_leaf <Payload> to denote a situation where <Type> and <Len> are still unexpanded, and <Payload> has been expanded a single time with the first option, producing byte sym_leaf whose value has not yet been determined. If the constraint set associated with the derivation tree is ever deemed unsatisfiable, we backtrack the last decision (in this case, to <Type> <Len> <Payload>). On the other hand, if the derivation tree has no more open nonterminals and the constraint set is satisfiable, we instantiate all the instances of sym_leaf with concrete values and output the solution. Intuitively, the problem is difficult because (i) the search space is over derivation trees of potentially unbounded size, and (ii) the search space is inherently both *syntactic* (over various derivations of the context-free aspect of the input grammar) and *semantic* (handling the annotated production rule constraints). We must craft the search algorithm to overcome a series of obstacles related to these key differences.

Derivation tree normalization. When constructing a candidate solution, some "decisions" are forced—namely, production rule expansions for nonterminals with only one production rule option, and type annotation expansions. To trim down the search space, we always normalize the candidate derivation tree by performing these forced expansions before making any decisions. Then, when backtracking, we always backtrack to a normalized derivation tree. In the working example, the search would start with derivation tree <Packet>. However, since <Packet> only has one expansion option, it takes a normalization step to <Type> <Len> <Payload>. But here, <Type> and <Len> also only have one expansion option, so we fully normalize to sym_leaf1 sym_leaf2 <Payload>. We cannot normalize any further, since <Payload> has more than one expansion option. Then, upon backtracking, we will never backtrack to <Packet>, since it is not in normal form.

Incremental solving. Two universally desirable qualities of search algorithms are (i) the ability to quickly identify when a candidate solution is infeasible ("fail fast"), and (ii) the ability to efficiently mend candidate solutions which are promising but imperfect. We achieve both of these qualities through an application of incremental SMT solving techniques. Intuitively, relevant semantic constraints are asserted as eagerly as possible to the solver instance. Whenever decisions are made (in this case, the choice of a production rule option to explore), an assertion level is pushed, and whenever a decision is backtracked, the assertion stack is popped, undoing the assertions associated with the corresponding decision. We achieve quality (i) because we can detect and backtrack as soon as the set of relevant constraints becomes unsatisfiable, rather than fully expanding a candidate derivation before realizing that it is an infeasible candidate. Similarly, we achieve quality (ii). Consider a situation where two decisions are made. Suppose each decision has an associated set of constraints, and the conjunction of all the constraints is satisfiable. After pushing the first set of constraints and querying for satisfiability, we *do not have to immediately instantiate a model into the candidate term.* Instead, we allow the partial solution to remain stored into the solver state, and we only retrieve a solution when all constraints are pushed. If we retrieve a model and instantiate it into the candidate term immediately, we risk unnecessarily stumbling into an unsatisfiable state. In comparison, prior work [30] instantiates models from SMT queries immediately into the candidate term. Incremental solving is possible in our approach because we iteratively build and backtrack a candidate solution, rather than performing non-chronological backtracking where the algorithm jumps between completely unrelated derivation trees [30]. For the working example, once we normalize the derivation tree to sym_leaf1 sym_leaf2 <Payload>, we eagerly assert all *applicable* semantic constraints we encountered during the expansion step. In this case, we assert <Type> = int_to_bv(8, 1) => <Payload>.<Byte> < 32 and <Type> = int_to_bv(8, 0) lor <Type> = int_to_bv(8, 1). But, we do not assert the constraint associated with <Len>, since it is a derived field. Also, we do not assert the constraint on line 6, since it has not been encountered yet. We check that the constraint set is satisfiable, but we refrain from instantiating specific values into the derivation tree.

IDS. For highly recursive grammars, even without context sensitivity, complete and terminating algorithms for the generation of members of the language is nontrivial. Unstrategic approaches may diverge or generate terms that are too large to handle once context-sensitive constraints are introduced. We employ a classic search strategy, iterative deepening search (IDS), which exhaustively explores the search space to a given depth limit before restarting and increasing the depth limit. Our formal approach is parametric with respect to the specific choice of production rule expansion to explore next. As much as possible, we separate the *core theory* of the approach from *specific strategies and implementation decisions*. In the working example, consider a depth limit of 1 with current derivation tree sym_leaf_1 sym_leaf_2 <Payload> (with depth 0). Say we expand <Payload> with the first option to sym_leaf_1 sym_leaf_2 sym_leaf_3 <Payload>, and then again to sym_leaf_1 sym_leaf_2 sym_leaf_3 sym_leaf_4 <Payload>. However, this pushes the current depth of 2 beyond the depth limit of 1. So, we backtrack the last decision to sym_leaf_1 sym_leaf_2 sym_leaf_3 <Payload>. But here, expanding to the second option (and normalizing) to sym_leaf_1 sym_leaf_2 sym_leaf_3 sym_leaf_4 still pushes beyond the depth limit, so we backtrack to sym_leaf_1 sym_leaf_2 <Payload> and instead expand (and normalize) to sym_leaf_1 sym_leaf_2 sym_leaf_3. Since we took the second expansion option at <Payload>, we assert (<Byte> bvand 0b11110000) = 0b10100000. Since the constraint set is satisfiable and there is nothing left to expand, we instantiate the model from the SMT solver into the derivation tree to give 0b00000000 sym_leaf2 0b00000000. Finally, computing the derived (length) field yields 0b00000000 0b00011000 0b00000000. If we had not found any solution at depth limit 1, we would have restarted and increased the depth limit to 2.

Multiple solutions. While we present our approach in terms of a search problem that seeks a single solution, it can be extended to produce multiple solutions. After finding a solution, we (i) pop to the zeroth assertion level, (ii) assert a blocking clause which prevents the same solution from being found multiple times, and (iii) push an assertion frame to start our next search from the first assertion level. With this strategy, we ensure that blocking clauses are never popped from the assertion stack.

Unsatisfiable inputs. Goblin will detect and report if it concludes that an input grammar is unsatisfiable (i.e., its language is empty). Goblin derives Unsat when the current constraint set is unsatisfiable, there is no decision to backtrack, and no candidate solutions have been discarded due to hitting the depth limit.

Goblin and SMT semantics. We use an incremental SMT solver as a backend reasoning engine. However, Goblin's semantics (described earlier in Section 3) are defined in terms of a satisfaction relation $\models_{\mathcal{G}}$ between abstract syntax trees and Goblin formulas, while SMT semantics are defined in terms of a satisfaction relation $\models_{\mathcal{T}}$ between first-order models and first-order logic formulas, modulo theories of interest. We bridge this gap in semantics by defining two functions, universalize_c and universalize_m, which translates Goblin constraints and models to SMT constraints and models. Intuitively, the universalization functions replace dot notation expressions with variables, where the variable name is produced by serializing the path from the root node to the corresponding leaf node in the candidate derivation tree. In the working example, the universalized version of Goblin constraint <Type> = int_to_bv(8, 1) => <Payload>.<Byte> < 32 is packet__type = int_to_bv(8, 1) => packet__payload__byte < 32.

6 Formal Approach

In this section, we present a formal description of the approach described in Section 3 by formulating it as a calculus. We present some preliminaries, give an intuitive description of the calculus, and then prove that the calculus offers certain guarantees, namely solution soundness, refutation soundness, and solution completeness.

Derivation trees. We define a **derivation tree** as an abstract syntax tree that is potentially unfinished. More precisely, the labeling function ℓ is updated to have type $V \longrightarrow N \cup (\bigcup \Lambda) \cup \{\text{None}\}$; case (c) of syntactic restriction (ii) is updated to allow $\ell(v) = \text{None}$; and we introduce a new case (d) of syntactic restriction (ii) which allows v to be a leaf vertex with $\ell(v) \in N$. We denote the language of derivation trees of a grammar G as $\mathcal{L}_{\text{DT}}(G)$, which is analogous to $\mathcal{L}_{\text{AST}}(G)$ but for derivation trees rather than ASTs.

Configurations. The calculus operates on **configurations**, which are either 7-tuples $(DT, \mathcal{O}, C, A, DS, \text{visited}, L)$ or the distinguished symbol $\bot$, denoting that there is no solution. Each rule of the calculus updates the current configuration to a new configuration, called a **conclusion**, given that the specified preconditions hold. In each rule, unprimed variables denote the current configuration, and the primed variables represent the updated configuration. If the primed version of some configuration variable is left undefined in the conclusion of a rule, it is assumed that the configuration variable is left unchanged. A rule **applies** to a configuration C if all the rule's preconditions hold for C. A configuration is considered **saturated** if it is not $\bot$ and no rule applies. The **initial configuration** for a given ACFG $G = (N, R, \Gamma, S)$ is the seven-tuple $(DT, \mathcal{O}, \{\}, \{\}, [], \{\}, L)$, where DT is the derivation containing only a single node labeled with S, and $\mathcal{O}$ is at the zeroth assertion level with no assertions pushed, and L is some natural number (initial depth limit).

Proof trees. A **proof tree** is a directed tree where every node maps to a configuration, and all the children of each node are obtained by the application of some applicable rule to the node. A proof tree T' **derives** from a proof tree T if T' is obtainable by applying a rule to one of T's leaves. A **derivation** is a finite or countably infinite sequence of proof trees such that each proof tree in the sequence derives from the previous proof tree and every proof tree has an initial configuration at the root. Then, a **refutation** is a finite derivation where all leaves of the final derivation tree are $\bot$, and a **solution** is a finite derivation where all leaves of the final derivation are saturated.

Note that since each rule in the calculus only produces a single resulting configuration, every proof tree is a path with a single leaf.

Configuration description. The configuration variables are now described more intuitively.

1. DT is a derivation tree that represents the (potentially unfinished and backtrackable) construction of an output term.
2. O represents an incremental SMT oracle, supporting operations push, pop, assert, and check_sat
3. C represents a set of relevant constraints (with respect to DT).
4. A represents the set of constraints that have been asserted since the last pop() of the incremental SMT oracle.
5. DS represents a decision stack of derivation trees, supporting operations push, pop, and is_empty.
6. visited :: $\mathcal{DT} \times V \to \mathcal{P}(R)$ represents a map of $\langle$ derivation tree, vertex $\rangle$ pairs to the set of production rule expansions that have been explored.
7. L is a natural number representing the current depth limit.

A configuration is **satisfiable** if DT, or some derivation tree in DS, can be extended to an AST in $\mathcal{L}_{\text{AST}}(G)$, and otherwise, it is **unsatisfiable**. Every AST in $\mathcal{L}_{\text{AST}}(G)$ that is an extension of DT or some derivation tree in DS, only using expansions not captured by visited, is called a **model** of the configuration.

Analogously, an ACFG G is called **satisfiable** if its language is nonempty, and otherwise it is **unsatisfiable**.

Informal rule explanations. Decide is a rule for expanding an open derivation tree at some vertex v associated with a nonterminal with more than one production rule option. We choose an arbitrary production rule option and perform the expansion. NormalizePR is a rule for normalizing an open derivation tree. It is similar to DecidePR, but for "forced" expansions where a given nonterminal has exactly one production rule option, so no choice needs to be made. NormalizeTA is analogous to NormalizePR, but for type annotations. Note that we never make decisions for type annotations, since there is only one expansion option (one child with a value of the corresponding type). Assert is a rule for asserting an applicable constraint at the current assertion level to the incremental SMT oracle. BacktrackDepth is a rule for reverting the last decision and backtracking when the depth limit has been reached at some open leaf. BacktrackUnsat is a rule for reverting the last decision and backtracking when the set of asserted constraints becomes unsatisfiable. RestartDepth and RestartUnsat are analogous to BacktrackDepth and BacktrackUnsat, except they apply when there is no prior decision to backtrack. Instead, there is a full restart, and the depth limit is increased. Solve is a rule for instantiating a finished derivation tree with concrete values, assuming the set of asserted constraints is satisfiable. Fail is a rule for giving up when the current set of asserted constraints is unsatisfiable, and there is no prior decision to backtrack. Moreover, the precondition on *bd?* enforces that we have not backtracked due to the depth limit (as opposed to backtracking due to an unsatisfiable constraint set) since the last restart.

Other preliminaries. The set of **active assertions** for O is defined as the set of assertions at the current, or lower, assertion levels of O. A constraint φ is considered **applicable** in a derivation tree dt if for all terms t in φ, $\mathcal{I}_{dt}(t) \neq \top$ (that is, every dot notation expression references children that are actually present in the derivation tree). We use $\mathcal{M}$ to denote a model returned by check_sat; open_leaves :: $\mathcal{DT} \to \mathcal{P}(V)$ returns the set of open leaves (that is, leaves with $\ell(v) \in N$) in the input derivation tree; applies :: $\mathcal{DT} \times \mathcal{C} \to$ Bool returns whether or not the input constraint applies to the input derivation tree; expand :: $\mathcal{DT} \times V \times N^* \to \mathcal{DT}$ expands the input derivation tree at the given vertex with the given child node labels; new_dt :: $N \to \mathcal{DT}$ takes a nonterminal n as input and produces a new derivation tree with a single vertex of label n; is_normalized :: $\mathcal{DT} \to$ Bool is shorthand for whether or not the input derivation is normalized (i.e., whether or not NormalizePr or NormalizeTa is applicable); universalize_c :: $\mathcal{DT} \times V \times \mathcal{C} \to \mathcal{C}$ rewrites the input constraint, phrasing the nonterminal expressions in terms of absolute paths from the root of the current derivation tree to vertex v; and universalize_m :: $\mathcal{DT} \to \mathfrak{M}$ translates a derivation tree to a first-order logic model by creating a variable and value for each path in the input derivation tree from root to leaf.

Lemmas. Before presenting the main results, we first present some useful helper lemmas.

First, we tie Goblin's semantics, defined in Section 3, with SMT semantics to soundly enable reasoning with SMT solvers, relating Goblin's satisfaction relation $\models_{\mathcal{G}}$ with SMT satisfaction, denoted $\models_{\mathcal{T}}$ (first-order logic satisfaction with respect to any relevant theories). This gap is bridged by the universalize_m and universalize_c functions, which translate derivation trees to SMT models and Goblin constraints to SMT constraints, respectively. Inuitively, the functions perform a *flattening* where each path in the derivation tree from root to leaf gets a corresponding SMT variable. Note that the function and predicate symbols are directly translated—the functions and predicates in Goblin's constraint language have direct SMT-LIB [6] analogues.

$$
\begin{aligned}
&\text{universalize\_c}(dt, v, \varphi_1 \wedge \varphi_2) = \\
&\qquad \text{universalize\_c}(dt, v, \varphi_1) \wedge \text{universalize\_c}(dt, v, \varphi_2) \\
&\text{universalize\_c}(dt, v, \neg\varphi) = \\
&\qquad \neg\text{universalize\_c}(dt, v, \varphi) \\
&\text{universalize\_c}(dt, v, p(t_1, \ldots, t_n)) = \\
&\qquad p(\text{universalize\_c}(dt, v, t_1), \ldots, \text{universalize\_c}(dt, v, t_n)) \\
&\text{universalize\_c}(dt, v, f(t_1, \ldots, t_n)) = \\
&\qquad f(\text{universalize\_c}(dt, v, t_1), \ldots, \text{universalize\_c}(dt, v, t_n)) \\
&\text{universalize\_c}(dt, v, \langle \text{nt}_1 \rangle.\langle \text{nt}_2 \rangle.\ldots.\langle \text{nt}_n \rangle) = \\
&\qquad \langle \ell(\text{root}(dt)) \rangle\_\ldots\_\langle \ell(\text{v}) \rangle\_\langle \text{nt}_1 \rangle\_\langle \text{nt}_2 \rangle\_\ldots\_\langle \text{nt}_n \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\text{universalize\_m}(dt) = \{(p, v) \mid \\
&\quad l \in \text{leaves}(dt) \wedge p = \langle \ell(\text{root(dt)}) \rangle\_\ldots\_\langle \ell(\text{parent}(l)) \rangle \wedge v = \ell(l)\}
\end{aligned}
$$

$$\textsc{Decide}\ \frac{\begin{array}{c}\text{is\_normalized}(DT)\quad \text{depth}(DT)\le L\quad v\in\text{open\_leaves}(DT)\quad \ell(v),NTs,D\notin\text{visited}(DT,v)\\ \{\varphi\in C\mid \text{applies}(DT,\varphi)\wedge\neg(\varphi\in A)\}=\emptyset\quad D^*=\text{universalize\_c}(DT,v,D)\end{array}}{\begin{array}{c}DT'\leftarrow\text{expand}(DT,v,NTs)\quad C'\leftarrow C\cup D^*\quad \text{visited}'\leftarrow\text{add}(\text{visited},(DT,v),(\ell(v),NTs,D))\\ DS'\leftarrow DT::DS\quad O'\leftarrow\text{push}(O)\quad \text{depth}'\leftarrow\text{depth}[\text{DT}'\mapsto\text{depth}(DT)+1]\end{array}}$$

$$\textsc{NormalizePR}\ \frac{v\in\text{open\_leaves}(DT)\quad \text{depth}(DT)\le L\quad |\{\ell(v),NTs,D\in R\}|=1\quad D^*=\text{universalize\_c}(DT,v,D)}{DT'\leftarrow\text{expand}(DT,v,NTs)\quad C'\leftarrow C\cup D^*}$$

$$\textsc{NormalizeTA}\ \frac{v\in\text{open\_leaves}(DT)\quad \text{depth}(DT)\le L\quad \ell(v),\tau\in\Gamma}{DT'\leftarrow\text{expand}(DT,v,[\,])}\qquad\qquad \textsc{Assert}\ \frac{\varphi\in C\quad \varphi\notin A\quad \text{applies}(DT,\varphi)}{O'\leftarrow\text{assert}(O,\{\varphi\})\quad A'\leftarrow A\cup\{\varphi\}}$$

$$\textsc{BacktrackUnsat}\ \frac{\text{check\_sat}(O)=\textsc{Unsat}\quad DS=dt::DS_2\quad V_b=\text{vertices}(DT)-\text{vertices}(dt)}{\begin{array}{c}DT'\leftarrow dt\quad DS'\leftarrow DS_2\quad O'\leftarrow\text{pop}(O,1)\quad A'\leftarrow\emptyset\\ C'\leftarrow C-\{\text{universalize\_c}(DT,v',\varphi)\mid \ell(v'),\_,\varphi\in R\wedge v'\in V_b\}\end{array}}$$

$$\textsc{BacktrackDepth}\ \frac{\text{depth}(DT)>L\quad DS=dt::DS_2\quad V_b=\text{vertices}(DT)-\text{vertices}(dt)}{\begin{array}{c}DT'\leftarrow dt\quad DS'\leftarrow DS_2\quad O'\leftarrow\text{pop}(O,1)\quad bd?'\leftarrow\texttt{true}\quad A'\leftarrow\emptyset\\ C'\leftarrow C-\{\text{universalize\_c}(DT,v',\varphi)\mid \ell(v'),\_,\varphi\in R\wedge v'\in V_b\}\end{array}}$$

$$\textsc{Solve}\ \frac{\text{open\_leaves}(DT)=\emptyset\quad \{\varphi\in C\mid \text{applies}(DT,\varphi)\wedge\neg(\varphi\in A)\}=\emptyset\quad \text{get\_model}(O)=\mathcal{M}\quad \text{depth}(DT)\le L}{\mathcal{DT}'\leftarrow\{\text{instantiate}(DT,\mathcal{M})\}}$$

$$\textsc{RestartUnsat}\ \frac{\text{check\_sat}(O)=\textsc{Unsat}\quad DS=[\,]\quad bd?}{L'\leftarrow L+1\quad DT'\leftarrow\text{new\_dt}(\text{S})\quad O'\leftarrow\text{push}(\text{reset}(O))\quad C',A'\leftarrow\emptyset\quad bd?'\leftarrow\texttt{false}\quad \text{visited}'\leftarrow\{\}}$$

$$\textsc{RestartDepth}\ \frac{v\in\text{open\_leaves}(DT)\quad \text{depth}(DT)>L\quad DS=[\,]}{L'\leftarrow L+1\quad DT'\leftarrow\text{new\_dt}(\text{S})\quad O'\leftarrow\text{push}(\text{reset}(O))\quad C',A'\leftarrow\emptyset\quad bd?'\leftarrow\texttt{false}\quad \text{visited}'\leftarrow\{\}}$$

$$\textsc{Fail}\ \frac{\text{check\_sat}(O)=\textsc{Unsat}\quad DS=[\,]\quad \neg bd?}{\bot}$$

To tie the semantics of Goblin to the semantics of SMT, then intuitively, we must demonstrate that a Goblin constraint φ associated with some production rule in the derivation tree is satisfied by the tree exactly when some corresponding SMT model satisfies the corresponding SMT version of the constraint (where the corresponding SMT model and constraint are defined using universalize_m and universalize_c).

Lemma 6.1. *Consider an arbitrary Goblin derivation tree tr, which has an arbitrary applicable semantic constraint φ at some arbitrary vertex v. We denote the subtree of tr rooted at v by tr'. Then,*

$$tr' \models_{\mathcal{G}} \varphi \text{ iff } \text{universalize\_m}(tr) \models_{\mathcal{T}} \text{universalize\_c}(tr,v,\varphi).$$

Proof. Assume φ includes some arbitrary dot notation term t. We denote the Goblin interpretation of t in tr' by $\mathcal{I}_{tr'}(t)$ (as in Section 3), and analogously, we denote the SMT interpretation of the universalized version of t by $\mathcal{I}_{\text{universalize\_m}(tr)}(\text{universalize\_c}(tr,v,t))$. We argue that

$$\mathcal{I}_{tr'}(t)=\mathcal{I}_{\text{universalize\_m}(tr)}(\text{universalize\_c}(tr,v,t)).$$

The left-hand side is defined in the Goblin semantics as the value in tr' at the leaf node l found along the path (from root note v) denoted by t. The right-hand side is defined as the value of the variable whose name was constructed by serializing the path from the root of dt to v, and from v to the corresponding leaf node denoted by t. Notice that this leaf node is precisely the same as l, hence, the values are equal.

We argued that the interpretations assigned to dot notation terms is equivalent between Goblin semantics and SMT semantics (using our universalization functions). Besides dot notation, *for applicable constraints*, the satisfaction relations $\models_{\mathcal{G}}$ is defined equivalently to $\models_{\mathcal{T}}$. Hence, the claim holds. □

Using this lemma, we can identify derivation tree constraints taken from R with universalized SMT constraints produced by the universalize_c assumption (see Decide, NormalizePr, and NormalizeTa, which translate the derivation tree constraints to SMT constraints with universalize_c before adding them to the constraint set). Further, we can identify SMT models with derivation tree leaf values (see Solve, which instantiates the SMT model from O into DT). Note that with the universalization functions, we avoid name clashes between variables in semantic constraints associated with nonterminals that are explored multiple times in the same derivation.

Next, we present two lemmas to couple the state of O to DT in every configuration (intuitively, the assertions active in O are exactly those that are relevant to DT).

Lemma 6.2. *For every configuration of every derivation, every active assertion in O is applicable in DT.*

Proof. The relevant rules are those that introduce new assertions or shrink DT, namely, Assert, BacktrackUnsat, BacktrackDepth, RestartUnsat, and RestartDepth. We proceed by structural induction on the rules, mentioning only these relevant cases. (1) Assert only adds assertions that are applicable in DT from the third precondition. (2) BacktrackUnsat and BacktrackDepth both update DT to be the old head of DS and pop O. Now notice that the only rule which pushes to the decision stack or pushes an assertion level is Decide. By the inductive hypothesis, when DT was pushed to the decision stack by Decide, O's active assertions were all applicable to DT. Notice that every push to DS has a corresponding push to O (see Decide), and analogously for pops (see BacktrackDepth and BacktrackUnsat). Hence, in the backtrack rules, when we pop to retrieve DT from DS while simultaneously popping O, we return exactly to the same state of O from when DT was pushed. Therefore, all the asserted constraints are applicable. (3) RestartUnsat and RestartDepth both trivially satisfy the claim, as they reset the assertion stack (popping all assertions). □

Lemma 6.3. *For every configuration of every derivation, every constraint from the input grammar which is applicable in DT is in C.*

Proof. New expansions of DT involving new production rule constraints only occur in Decide and NormalizePr. In both cases, all constraints associated with the chosen production rule (including the applicable ones) are added to C. □

Lemma 6.4. *For input grammar G, for every configuration of every derivation, $DT \in \mathcal{L}_{DT}(G)$.*

Proof. By structural induction on the rules. The relevant cases are Decide, NormalizePr, and NormalizeTa. In Decide and NormalizePr, if $DT \in \mathcal{L}_{DT}(G)$, then $DT' \in \mathcal{L}_{DT}(G)$ since DT' is constructed by expanding DT according to one of the production rules in R. In NormalizeTa, if $DT \in \mathcal{L}_{DT}(G)$, then $DT' \in \mathcal{L}_{DT}(G)$ since DT is expanded by filling in a symbolic leaf value (i.e., an uninstantiated leaf with $\ell(v) = \text{None}$) for some vertex associated with a type annotation in Γ. The other cases do not expand DT, so we trivially get that $DT' \in \mathcal{L}_{DT}(G)$. □

Finally, we present lemmas that are directly relevant for refutation and solution soundness (respectively).

Lemma 6.5. *Every rule except for Solve and BacktrackDepth preserves models.*

Proof. By cases. (1) Decide updates DT by making a new decision and expanding an open leaf according to a production rule option in R. The models associated with DT can be partitioned into those that use this expansion for the open leaf, and those that use a different expansion. The prior set is still captured in DT', while the latter set are captured by the new head of DS', namely, DT. (2) NormalizePr and NormalizeTa both expand DT without altering the decision stack. Since the expansions are forced (a precondition is that there are no alternative expansions), they cannot lose models. (3) Assert does not directly alter DT or DS, so it only relevant through Lemmas 6.3 and 6.2, which are used in the Solve case. (4) BacktrackUnsat discards DT, transfers the head of DS to DT', and maintains the rest of DS. We know from Lemma 6.2 that all the current constraints are relevant for DT, and yet the constraint set is unsatisfiable. Hence, DT cannot be extended to any models, so it is safe to discard. Since the rest of the derivation trees in DS are shifted/maintained, no models are lost. (5) RestartUnsat removes everything but the root from DT, and it does not change DS, so it cannot possibly remove models. (6) RestartDepth proceeds by the same argument as RestartUnsat (7) For Fail, We know from Lemma 6.2 that all the current constraints are relevant for DT, and yet the constraint set is unsatisfiable. Hence, DT cannot be extended to any models. But further, DS is empty, so there cannot be any models in the unprimed configuration. Since there are no models in the unprimed configuration, losing models is impossible. □

Below, we use *productive* to denote that a nonterminal is capable of deriving some finite string.

Lemma 6.6. *For an input ACFG G such that every nonterminal is productive, if the calculus does not terminate at depth limit n, it enumerates all trees of depth $\leq n$ which could possibly extend to a model before moving on to depth $n + 1$.*

Proof. The calculus implements an iterative deepening search, which exhaustively explores the search space of derivation trees $\leq L$ using Decide to store decisions to be backtracked later, BacktrackDepth to enforce the depth limit, and RestartDepth to increase the depth limit once the current limit has been thoroughly explored.

First, we argue by cases that IDS always *makes progress* in the sense that it is impossible to have an infinite chain of rule applications that does not include Decide. Assert relies on applications of Decide to collect formulas in C to assert. BacktrackDepth and BacktrackUnsat rely on decisions to backtrack. Solve always produces a saturated conclusion (argued in Theorem 6.7). RestartUnsat relies on BacktrackDepth for the precondition involving *bd?*, which in turn relies again on Decide. RestartDepth, NormalizePr, and/or NormalizeTa could, in principle, chain indefinitely without allowing an application of Decide. However, this is impossible with our assumption that each nonterminal is productive.

Notice that calculus does not enumerate every possible derivation tree due to the presence of BacktrackUnsat and RestartUnsat, which discard derivation trees without exploring every possible expansion. However, whenever these rules are invoked, the constraint set is in an unsatisfiable state. Since the constraint set is applicable to DT by Lemma 6.2, no extension of DT could possibly lead to a model. Thus, it is safe to discard DT and refrain from exploring its possible extensions.

Since IDS always makes progress and all other derivation trees are enumerated by IDS, the claim holds. □

Theorems. We now present the main theorems.

Theorem 6.7. *(Refutation soundness) Every refutation has an initial configuration associated with an unsatisfiable grammar.*

Proof. First, we argue that no refutation can include an application of Solve. Assume for a contradiction that Solve is applied somewhere in a refutation. Then, the conclusion of this Solve rule must not be a saturated configuration, because the proof tree must have $\perp$ at every leaf. However, this is impossible, as after applying Solve, none of the other rules can be applicable: Decide, NormalizePr, and NormalizeTa require some open leaf, but one cannot exist (see the preconditions to Solve). Assert requires the existence of some $\varphi \in C$ such that φ is applicable but not in A, but this is also impossible due to a precondition of Solve. BacktrackUnsat requires that the current asserted constraint set by unsatisfiable, but this cannot be the case since O just returned a model from check_sat. BacktrackDepth requires the depth of DT to be above L, but this contradicts a precondition of Solve. RestartUnsat and RestartDepth proceed with the same arguments as BacktrackUnsat and BacktrackDepth. Fail proceeds with the same argument as BacktrackUnsat.

Second, we argue that every refutation containing BacktrackDepth can be rewritten a different refutation that does not contain BacktrackDepth. (We do this because BacktrackDepth does not necessarily preserve models.) Notice that due to the precondition on *bd?*, no proof tree can include an application of BacktrackDepth, followed by an application of Fail, without an application of RestartDepth or RestartUnsat in between. Therefore, consider an arbitrary proof tree containing BacktrackDepth. Then, consider the last application of RestartDepth or RestartUnsat, and take the subtree rooted at the conclusion of this application as the new refutation. (This conclusion is a valid initial configuration, just with a higher depth limit.)

Since Solve and BacktrackDepth are never present, we can argue from structural induction on proof trees using Lemma 6.5 that any proof tree with a leaf containing $\perp$ must also have an unsatisfiable root. ◻

Theorem 6.8. *(Solution soundness) Every solution has an initial configuration associated with a satisfiable grammar.*

Proof. We argue that every saturated leaf of a solution corresponds to a model of the input grammar. The arguments proceeds in three steps. We argue that (1) the conclusion of an application to Solve must be saturated, (2) the conclusion of an application of Solve corresponds to a model, and (3) the conclusion of any rule except for Solve cannot be saturated.

We already argued step (1) in Theorem 6.7.

For step (2), we can directly take DT' as the model. We know DT' is syntactically valid from Lemma 6.4, and we know it is semantically valid with respect to all the applicable constraints from Lemmas 6.3, 6.2, and 6.1, along with the precondition that all applicable constraints in C must be asserted since the last pop (second precondition). Finally, the inapplicable constraints are trivially satisfied, according to the semantics in Section 3.

We argue step (3) by contradiction. Assume the conclusion of another rule is saturated. First, the depth of DT must be $\leq L$, or else BacktrackDepth or RestartDepth would be applicable. Second, the derivation tree cannot have any open leaves, or else Decide, NormalizePr, or NormalizeTa would be applicable (assuming DT has depth $\leq L$). Also, the derivation tree cannot have any applicable but unasserted constraints, or else Assert would be applicable. Finally, the current constraint set cannot be unsatisfiable, or else BacktrackUnsat or RestartUnsat would be applicable. However, since all of the above hold, it must be the case that Solve is applicable, a contradiction with the assumption that the conclusion was saturated. ◻

In solution completeness, note that the precondition of nonterminal productivity is enforced in Goblin by a syntactic check before the main solve algorithm.

Theorem 6.9. *(Solution completeness) For every satisfiable grammar G such that every nonterminal is productive, there exists a solution with an initial configuration built from G.*

Proof. The result directly follows from Lemma 6.6. ◻

Notably, the calculus is refutationally incomplete in the sense that there might not exist a refutation for unsatisfiable initial configurations. This is unsurprising, as the unsatisfiability of the initial configuration may be an inductive property (e.g., a grammar denoting a list of integers whose individual elements are negative but whose total sum must be positive). Refutations are not discussed in prior work [30], so the presence of refutations and refutation soundness in our calculus is an improvement over prior work, despite the lack of refutation completeness.

7 CLP Semantics

We present an alternative language semantics for Goblin as a *translational semantics* to CLP problems. While the semantics described in Section 3 works naturally with our proofs, the CLP semantics is also useful to get a sense for how Goblin relates to CLP. More concretely, we define a translation of an input ACFG to a CLP problem, which we model as a set of constrained horn clauses (CHCs). Each CHC is of the form $H \leftarrow B_1, \ldots, B_n, \phi$, where H and B_1 through B_n are terms, and ϕ is a constraint over the free variables from the prior terms. Assuming the existence of a CLP solver that can handle all the constraints in the generated CLP problem, the CLP semantics can be viewed as a path toward an alternative, CLP-based engine for Goblin.

We have

$$[\![G]\!] = \bigcup_{nt,N,C \in R} [\![nt, N, C]\!]_{\mathrm{pr}} \cup \bigcup_{nt,\tau \in \Gamma} [\![nt, \tau]\!]_{\mathrm{ta}},$$

where

$$
\begin{aligned}
[\![nt, \tau]\!]_{\mathsf{ta}} &= \{nt(NT) \leftarrow NT \in \tau\} \\
[\![nt, N, C]\!]_{\mathsf{pr}} &= \{[\![nt, N, C]\!]_{\mathsf{lhs}} \leftarrow [\![nt, N, C]\!]_{\mathsf{rhs}}\} \\
&\quad \cup \bigcup_{e \in \mathsf{nt\_exprs}(C)} [\![e]\!]_{\mathsf{fe}} \\
[\![nt, N, C]\!]_{\mathsf{lhs}} &= nt(nt_i(N_1, \ldots, N_m)), \text{ where each } N_j \in N \\
&\quad \text{for the } i\text{th production rule option for } nt \\
[\![nt, N, C]\!]_{\mathsf{rhs}} &= \bigwedge_{m \in N} m(M_i) \wedge \bigwedge_{e \in \mathsf{nt\_exprs}(C)} [\![e]\!]_{\mathsf{nt\_expr}} \wedge [\![C]\!]_{\mathsf{b\_expr}} \\
&\quad \text{for the } i\text{th occurrence of } m \text{ in } N \\
[\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe}} &= \{[\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe\_lhs}} \leftarrow [\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe\_rhs}}\} \\
&\quad \text{for each of } nt_1\text{'s production rules} \\
&\quad \cup [\![nt_2 \ldots nt_n]\!]_{\mathsf{fe}} \\
[\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe\_lhs}} &= nt_1(nt_{1,i}(N_1, \ldots, N_m), nt_n), \text{ where each } N_j \in N \\
&\quad \text{for the } i\text{th production rule option for } nt_1 \\
[\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe\_rhs}} &= \bigwedge [\![nt_2 \ldots nt_n]\!]_{\mathsf{nt\_expr}} \\
&\quad \wedge \mathsf{append}(\mathsf{nt_{n\_1}}, \ldots, \mathsf{nt_{n\_i}}) \\
[\![nt_1.nt_2]\!]_{\mathsf{fe}} &= \mathsf{nt_1\_nt_2}(\mathsf{nt}_{1,i}(q_1, \ldots, q_n), [NT_{2,1}, \ldots, NT_{2,m}]) \\
&\quad \text{for the } i\text{th production rule option for } \mathsf{nt}_1\text{, where} \\
&\quad \text{each } q_j \text{ is } \mathsf{NT}_{2,k} \text{ if the } j\text{th nonterminal in the } i\text{th} \\
&\quad \text{production rule for } \mathsf{nt}_1 \text{ is } \mathsf{nt}_2\text{, and \_ otherwise,} \\
&\quad \text{with } k \text{ ranging from 1 to } m \\
[\![nt_1 \ldots nt_n]\!]_{\mathsf{nt\_expr}} &= nt_1\_\ldots\_nt_n(NT_{1,i}, NT_1\_\ldots\_NT_{n,i}) \\
[\![C]\!]_{\mathsf{b\_expr}} &= \bigwedge_{c_i \in \mathsf{generalize}(C)} c_i
\end{aligned}
$$

Above, generalize performs the disambiguation of dot notation constraints analogous to resolve as discussed in Section 4, and nt_exprs(.) retrieves a set of all dot notation expressions present in the input constraints. Also, we use square brackets to denote list literals and append to denote a multi-arity list append function. The pattern "_" denotes a universal match for an unused value. In the encoding, we use both lowercase and uppercase names, which tie to corresponding nonterminals in the input grammar. By convention, we use the lowercase names to denote constructor symbols and the uppercase names to denote variables (e.g. nt and NT, both referring to the same nonterminal in the input grammar).

Intuitively, we partition the CHCs into those we need for type annotations and those we need for production rules. The type annotation CHCs simply state that the given nonterminal must be a member of its given type; the production rule CHCs are more complicated. First, the production rule CHC definitions are split into the left-hand side (LHS) and right-hand side (RHS). They use CLP terms to capture the structure of the grammar and CLP constraints to capture the semantic constraints from C. In addition to these LHS and RHS definitions, each production rule also generates CHCs which we call *field extractors*, defined by $[\![.]\!]_{\mathsf{fe}}$, which enable passing variables between clauses to support the dot notation constraints. The field extractor CHCs are invoked by calls defined by $[\![.]\!]_{\mathsf{nt\_expr}}$.

8 Implementation and Evaluation

Goblin Implementation. We implemented Goblin in ~9.9K lines of OCaml 5.1.1 code. For our underlying constraint solver, we use and fully support cvc5 [5] version 1.3.0. The main pipeline is depicted in Figure 1.

Code availability. Goblin's source code and experimental evaluation details are available at [3].

Empirical Evaluation. We evaluate Goblin on four case studies. The first three case studies, XML, CSV, and Scriptsize C input generation, serve as a direct comparison to prior work [30] and demonstrate Goblin's capability of handling complex constraints. In these case studies, we evaluate the correctness, efficiency, diversity, and length of generated inputs. The fourth case study involves generation of valid WiFi SAE commit and confirm frames, which cannot be handled directly by prior work due to the presence of rich bitvector constraints. Rather than taking a fixed grammar and constraints (as performed by prior work and in the prior two case studies), we demonstrate how Goblin's local production rule constraints can be easily integrated into a fuzzing workflow involving *grammar-level production rule mutation operations.*

8.1 XML, CSV, and Scriptsize C

We present our main results for the XML, CSV, and Scriptsize C case studies in Table 1. We collected the results with 1-hour runs of Goblin and ISLa locally on a Macbook Pro with an M2 chip and 16GB RAM. Our main research questions are:

RQ1. How *efficient* is Goblin compared to ISLa?
RQ2. How *diverse* are Goblin's outputs, compared to ISLa?
RQ3. What is the *size* of Goblin's outputs, compared to ISLa?

Both case studies involve *structural constraints.* In XML, there is a **definition before use** constraint that mandates that XML namespaces must be used only in scopes in which they are defined, a **no redefinition** constraint that mandates that XML namespaces are not redefined, and a **matching tags** constraint that matching closing and opening tags have the same identifiers. Scriptsize C has analogous definition before use and no redefinition constraints. In CSV, there is a **column count** constraint that mandates that all rows in the CSV file contain the same number of columns. In ISLa, structural constraints are supported through built-in **structural predicates**. However, in Goblin's DSL, structural predicates are not built-in. *Nonetheless, these constraints can still be encoded in Goblin.* (The encoding will be discussed in Section 8.2.)

Correctness. Based on our formal soundness guarantees, we can confidently claim that all generated inputs are correct with respect to the input specification. Prior work [2, 30] use *precision* as a metric for evaluation, where precision denotes the percentage of inputs successfully parsed by the system under test. We argue this metric is not relevant, since it is a metric of the system under test rather than of the input generator. However, we note that for CSV and XML, we formulated constraints equivalent to those in prior work.

Efficiency. To address RQ1, we compare the efficiency of Goblin to ISLa in terms of the number of inputs generated per second. Goblin outperforms ISLa by a wide margin for XML inputs and by a narrow margin for Scriptsize C inputs. We hypothesize that incremental solving boosts performance by allowing quicker backtracking and enabling the mending of partial solutions. Notably, Goblin struggles on CSV inputs. The semantic constraint, which mandates that all rows have the same number of columns, entirely restricts the *derivation strategy* at the context-free level, rather than the values of leaf terms. A current weakness of Goblin is that it does not use any special heuristic for choosing production rule expansions (only coin flips), which we believe explains the poor performance compared to ISLa.

Diversity. To address RQ2, we compare efficiency based on *accumulated k-path coverage* [17], which measures how many paths of length k from the grammar are actually present in the generated inputs, where a *path* refers to a sequence of nonterminal or terminal symbols. A higher k-path coverage is more diverse and thus better. As in [30], we present results with $k = 3$. Goblin performs very well, reaching near full coverage on every case study. However, we note that because Goblin generates ADT terms rather than strings, we do not use any terminal symbols in our encoding (terminal symbols are generated from leaf nonterminals, optionally with semantic constraints), which differs from ISLa's encoding which includes terminal symbols. So, although Goblin performs well, we cannot claim that the comparison with ISLa is direct.

Length. To address RQ2, we compare the mean length of inputs generated by Goblin and ISLa. Goblin's inputs, on average, are more concise than ISLa's. We report this metric to be informative, but we argue that there is no clear benefit or drawback to having shorter or longer inputs. Further, one can always impose additional semantic constraints or grammar-level mutations to either increase or decrease the length of generated inputs.

8.2 Structural Constraint Encoding

It may be unclear how one would express structural constraints in Goblin. Intuitively, we use local production rule constraints, along with dot notation and support for rich data types (e.g., sets), to specify the desired properties akin to how one would do so with an attribute grammar or a CLP system. The encodings are available along with Goblin's source code at [3]. As a simpler example, consider a grammar encoding two lists with the structural constraint that every element in the second list must also be present in the first:

Listing 11: Structural constraint

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

We specify the constraint by introducing extra nonterminal symbols into the grammar, preceded by underscores, to serve as means of passing context in the style of attribute grammars. We remind the reader that Goblin is designed as a minimal intermediate language, and a surface-level language with proper support for attributes and other features is left for future work.

Tool	Subject	Constraints	Efficiency (inputs/minute)	k-Path coverage	Mean length
Goblin	C	def-use; redef;	108.53	95.4	18.75
ISLa	C	def-use; redef;	101.70	50.72	24.01
Goblin	XML	def-use; redef; tags	100.00	84.44	11.71
ISLa	XML	def-use; redef; tags	48.00	75.37	38.03
Goblin	CSV	col count	19.5	100.00	88.45
ISLa	CSV	col count	91.83	100.00	487.08

Table 1: Results for each subject, tool, and associated metrics.

8.3 WiFi: WPA3-SAE Handshake Frames

We now present the results of our WiFi authentication case study. We focus on the WPA3-SAE handshake as this is the state-of-the-art WiFi authentication protocol [20] with complex packet formats consisting of (1) rich bitvector constraints, (2) inter-field dependencies, and (3) non-linear fields (cryptographic elements). We instantiate Goblin with an existing grammar-based black-box fuzzer Saecred [12] that generates context-sensitive SAE commit and confirm frames. Saecred reduces input generation to a SyGuS problem and uses a SyGuS solver [1] to generate byte sequences satisfying the grammar constraints. To evaluate Goblin, we replace Saecred's input generator with Goblin (referred to as Goblin-Saecred) and compare it with the original Saecred fuzzer.

Evaluation setup. We run Goblin-Saecred and Saecred for 24 hours each on an Intel Core Ultra 9 185H with 64 GB RAM. The black-box target was `hostapd` v2.10, a popular open-source WiFi access point software. We use the same grammar and constraints for both fuzzers, which we obtained from the Saecred authors.

Evaluation criteria. We propose three evaluation criteria: (1) *performance* – we measure the time taken for Goblin and SyGuS to generate byte sequences from the mutated grammar, (2) *expressiveness* – we measure the number of grammar-level mutations Goblin and SyGuS can handle, and (3) *effectiveness* – we measure the number of bugs uncovered in `hostapd` by the two fuzzers.

Performance. Both variations of Saecred were able to generate 97, 182 mutated grammars for byte serialization. Out of those, 49, 460 passed the Goblin language-emptiness checks. Figure 2 shows the (log-scaled) smoothed time taken by Goblin-Saecred and SyGuS-Saecred to generate byte sequences from the 49, 460 mutated grammar instances. Goblin-Saecred outperforms SyGuS-Saecred by a wide margin. Table 2 shows the summary statistics of the time taken to generate the byte sequences. Goblin-Saecred is 36x faster on average than SyGuS-Saecred.

Figure 2: Performance comparison of GOBLIN-SAECRED and SyGus-SAECRED on WiFi SAE commit and confirm frames. The x-axis shows the grammar instance, and the y-axis shows the log-scale time taken to produce byte serializations.

Metric / Method	Goblin	SyGuS	Unit
Successes	49,460	23,904	count
Failures	47,722	73,278	count
Total Grammars: 97,182			
Mean Time	0.0082	0.2968	seconds
Median Time	0.0074	0.2014	seconds
Std. Dev. Time	0.0588	0.3606	seconds

Table 2: Comparison of GOBLIN-SAECRED and SyGuS-SAECRED in terms of success/failure count and execution time. Goblin on average performs 36x faster than SyGuS.

Expressiveness. Out of the 97, 182 mutated grammar instances, some instances were unsolvable due to the presence of unsatisfiable constraints; however, the expressive capabilities of both approaches were vastly different. Table 2 shows that GOBLIN-SAECRED was able to handle 49, 460 grammar instances whereas SyGuS-SAECRED was only able to handle 23, 904 grammar instances, leading to 73, 278 failures as opposed to GOBLIN-SAECRED's 47, 722. This shows that GOBLIN is able to handle a wider variety of grammar-level mutations than SyGuS-based input generator of SAECRED.

Effectiveness. The 24 hour fuzzing campaigns resulted in GOBLIN-SAECRED uncovering all 4 known bugs from the original SAECRED paper [12], including denial-of-service and downgrade attacks.

9 Related Work

The most relevant related work was discussed in Sections 1 and 2: context-free grammar-based fuzzing [18], ISLa and Fandango [2, 30], system-specific input generators [8, 33], and CLP approaches [14].

SyGuS. Another related formalism is syntax-guided synthesis (SyGuS) [1], which presents a synthesis problem in terms of a function to synthesize that must be in the language of some user-defined grammar while also satisfying user-defined semantic constraints. In principle, one could use a SyGuS engine (e.g., cvc5SY [29]) as the backend reasoner for a tool like GOBLIN (as in SAECRED [12]). But, there are notable obstacles. For recursive grammars, in order to specify that a constraint applies to *all* instances of a production rule expansion during a derivation, the corresponding SyGuS constraint must be expressed as a recursive function, which is nontrivial to synthesize and often suffers from performance problems in a SyGuS setting. Further, by default, SyGuS solvers only support semantic constraints from the top level (i.e., predicates of type $\tau \rightarrow$ Bool, where τ is the type of the function to synthesize). Hence, supporting local production rule semantic constraints in general is nontrivial.

Attribute grammars. Attribute grammars [22] are a formalism built on top of context-free grammars, where the grammar can additionally assign *attributes* to each production rule. Attributes are additional variables that can be used to communicate context-sensitive information throughout a parse tree, and can be defined in terms of descendants (called *synthesized attributes*) or ancestors (called *inherited attributes*). One can use attributes to express and check semantic properties; however, prior work (e.g., [31]) on attribute grammars has focused on *parsing* rather than *generation*.

Property-based testing. Property-based testing (PBT), pioneered by QuickCheck [9], is a testing strategy in which the programmer annotates expected invariants in the source code, and a testing tool generates inputs to check whether or not the invariants are ever falsified. These approaches differ from the one employed by GOBLIN in two main ways: (*i*) GOBLIN operates with system-level input rather that internal inputs, and (*ii*) GOBLIN allows for the annotation of rich semantic constraints and employs heuristics that are more in depth than *generate-and-test* approaches typical in PBT.

Context-sensitive parsing. In general, most work on grammars with context-sensitive constraints has focused on parsing rather than input generation. For example, Parsley [24] is a tool that uses an input language similar in spirit to GOBLIN's, but it is a tool for synthesizing verified parsers for languages with context-sensitive constraints (rather than for input generation).

10 Discussion

Limitations. Compared to prior works, ours is most similar in spirit to ISLa [30], and we experience a similar set of tradeoffs. For non-derived fields, the constraint language space is restricted to being easily translatable to SMT-LIB [6]. Moreover, constraint solving can be expensive for some inputs.

11 Conclusion

We presented GOBLIN, a system for generating context-sensitive inputs. Compared to prior work, GOBLIN is novel in its capability to handle constraints from arbitrary SMT theories, its generality despite using a minimal core language, and its amenability to integration with grammar-based fuzzing workflows due to its semantics of local production rule constraints.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.

[2] JOSÉ ANTONIO ZAMUDIO AMAYA, MARIUS SMYTZEK, and ANDREAS ZELLER. 2025. FANDANGO: Evolving Language-Based Testing. (2025).

[3] Anonymous. 2025. Parse this! Summoning Context-Sensitive Inputs with Goblin. https://anonymous.4open.science/r/goblin_anonymous-44E1. Anonymous submission.

[4] John W Backus. 1959. The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *ICIP Proceedings*. 125–132.
[5] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.
[6] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.
[7] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of model checking*. Springer, 305–343.
[8] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 114–129.
[9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.
[10] Bruno Courcelle and Paul Franchi-Zannettacci. 1982. Attribute grammars and recursive program schemes I. *Theoretical Computer Science* 17, 2 (1982), 163–191.
[11] cvc5 Developers. 2025. cvc5 Nonstandard Theories. https://cvc5.github.io/docs/cvc5-1.0.2/theories/theories.html#non-standard-or-extended-theories.
[12] Muhammad Daniyal Pirwani Dar, Rob Lorch, Aliakbar Sadeghi, Vincenzo Sorcigli, Héloïse Gollier, Cesare Tinelli, Mathy Vanhoef, and Omar Chowdhury. 2025. Saecred: A State-Aware, Over-the-Air Protocol Testing Approach for Discovering Parsing Bugs in SAE Handshake Implementations of COTS Wi-Fi Access Points. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3691–3709.
[13] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-Proving. *Commun. ACM* 5, 7 (1962), 394–397. doi:10.1145/368273.368557
[14] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 725–730.
[15] Matheus E Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. 2022. {BrakTooth}: Causing havoc on bluetooth link manager via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 1025–1042.
[16] Matheus E Garbelini, Chundong Wang, and Sudipta Chattopadhyay. 2020. Greyhound: Directed greybox wi-fi fuzzing. *IEEE Transactions on Dependable and Secure Computing* 19, 2 (2020), 817–834.
[17] Nikolas Havrikov and Andreas Zeller. 2019. Systematically covering input structure. In *2019 34th IEEE/ACM international conference on automated software engineering (ASE)*. IEEE, 189–199.
[18] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458.
[19] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.
[20] IEEE Std 802.11. 2020. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec.
[21] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. 2020. ATFuzzer: dynamic analysis framework of AT interface for Android smartphones. *Digital Threats: Research and Practice* 1, 4 (2020), 1–29.
[22] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.
[23] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.
[24] Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. 2020. Research Report: The Parsley Data Format Definition Language. In 2020 IEEE Security and Privacy Workshops (SPW). 300–307.
[25] NIST. 2023. CVE-2021-1903. https://nvd.nist.gov/vuln/detail/cve-2023-29548.
[26] NIST. 2023. CVE-2021-30247. https://nvd.nist.gov/vuln/detail/cve-2021-30247.
[27] NIST. 2023. CVE-2023-29548. https://nvd.nist.gov/vuln/detail/cve-2023-29548.
[28] NIST. 2024. CVE-2024-52924. https://nvd.nist.gov/vuln/detail/CVE-2024-52924.
[29] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *International Conference on Computer Aided Verification*. Springer, 74–83.
[30] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 583–594.
[31] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Science of Computer Programming* 75, 1-2 (2010), 39–54.
[32] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.
[33] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.
[34] Michal Zalewski. 2014. American Fuzzy Lop (AFL). In *USENIX WOOT*. https://lcamtuf.coredump.cx/afl/.
[35] Jialun Zhang, Greg Morrisett, and Gang Tan. 2023. Interval parsing grammars for file format parsing. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1073–1095.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009