Verified ALL(*) Parsing with Semantic Actions and Dynamic Input Validation

Sam Lasser[1], Chris Casinghino[2], Derek Egolf[3],
Kathleen Fisher[4], and Cody Roux[5]

[1] Draper, Cambridge, USA
slasser@draper.com
[2] Jane Street, New York, USA
ccasinghino@janestreet.com
[3] Northeastern University, Boston, USA
egolf.d@northeastern.edu
[4] Tufts University, Medford, USA
kfisher@eecs.tufts.edu
[5] Amazon Web Services, Cambridge, USA
codyroux@amazon.com

Abstract. Formally verified parsers are powerful tools for preventing the kinds of errors that result from ad hoc parsing and validation of program input. However, verified parsers are often based on formalisms that are not expressive enough to capture the full definition of valid input for a given application. Specifications of many real-world data formats include both a syntactic component and one or more non-context-free semantic properties that a well-formed instance of the format must exhibit. A parser for context-free grammars (CFGs) cannot determine on its own whether an input is valid according to such a specification; it must be supplemented with additional validation checks.

In this work, we present CoStar++, a verified parser interpreter with semantic features that make it highly expressive in terms of both the language specifications it accepts and its output type. CoStar++ provides support for semantic predicates, enabling the user to write semantically rich grammars that include non-context-free properties. The interpreter also supports semantic actions that convert sequential inputs to structured outputs in a principled way. CoStar++ is implemented and verified with the Coq Proof Assistant, and it is based on the ALL(*) parsing algorithm. For all CFGs without left recursion, the interpreter is provably sound, complete, and terminating with respect to a semantic specification that takes predicates and actions into account. CoStar++ runs in linear time on benchmarks for four real-world data formats, three of which have non-context-free specifications.

Keywords: parsing · semantic actions · interactive theorem proving

1 Introduction

The term “shotgun parsing” refers to a programming antipattern in which code for parsing and validating input is interspersed with application code for processing that input. Proponents of high-assurance software argue for the use of dedicated parsing tools as an antidote to this fundamentally insecure practice [12]. Such parsers enable the user to write a declarative specification (e.g., a grammar) that describes the structure of valid input, and they reject inputs that do not match the specification, ensuring that only valid inputs reach the downstream application code. Formally verified parsers offer even greater security to the applications that rely on them. Verification techniques can provide strong guarantees that a parser accepts all and only the inputs that are valid according to the user's specification.

However, dedicated parsing tools are not always expressive enough to capture the full definition of valid input. For many real applications, the input specification includes both a context-free *syntactic* component and non-context-free *semantic* properties; in such a case, a parser for context-free grammars (CFGs) provides limited value. For example, a CFG can represent the syntax of valid XML, but it cannot capture the requirement that names in corresponding start and end tags must match (assuming that the set of names is infinite). Similarly, the syntactic specification for JSON is context-free, but some applications impose the additional requirement that JSON objects (collections of key-value pairs) contain no duplicate keys. Data dependencies are another common type of non-context-free property; many packet formats have a "tag-length-value" structure in which a length field indicates the size of the packet's data field. In each of these cases, a CFG-based parser is an incomplete substitute for shotgun parsing because it cannot enforce the semantic component of the input specification.

In this work, we present CoStar++, a verified parser interpreter[1] with two features—semantic predicates and semantic actions—that enable it to capture semantically rich specifications like those described above. Predicates enable the user to write input specifications that include non-context-free semantic properties. The interpreter checks these properties at runtime, ensuring that its output is well-formed. Actions give the user fine-grained control over the interpreter's output type. Actions also play an important role in supporting predicates; the interpreter must produce values with an expressive type in order to check interesting properties of those values. CoStar++ builds on the CoStar parser interpreter [11]. Like its predecessor, CoStar++ is based on the ALL(*) parsing algorithm, and it is implemented and verified with the Coq Proof Assistant.

Extending CoStar with predicates and actions gives rise to several challenges. CoStar is guaranteed to detect syntactically ambiguous inputs (inputs with more than one parse tree). In a semantic setting, the definition of ambiguity is more complex; it can be syntactic (multiple parse trees for an input) or semantic (multiple semantic values). In addition, it is not always possible to infer one kind of ambiguity from the other, because two parse trees can correspond to (a) two semantic values, (b) a single semantic value when the semantic actions for the two derivations produce the same value, or (c) no semantic value at all

[1] We use the term "parser interpreter" instead of "parser generator" because CoStar++ does not generate source code from a grammar; it converts a grammar to an in-memory data structure that a generic driver interprets at parse time.

when predicates fail during the semantic derivations! Finally, detecting semantic ambiguity is undecidable in the general case where semantic values do not have decidable equality, and we choose not to require this property so that the interpreter can produce incomparable values such as functions. However, it is still possible to detect the *absence* of semantic ambiguity. In the current work, we modify the CoStar ambiguity detection mechanism so that CoStar++ detects uniquely correct semantic values, and it detects syntactic ambiguity in the cases where semantic ambiguity is undecidable.

A second challenge is that ALL(*) as originally described [14] and as implemented by CoStar is incomplete with respect to the CoStar++ semantic specification. ALL(*) is a predictive parsing algorithm; at decision points, it nondeterministically explores possible paths until it identifies a uniquely viable path. This prediction strategy does not speculatively execute semantic actions or evaluate semantic predicates over those actions, for both efficiency and correctness reasons (the actions could alter mutable state in ways that cannot be undone). While this choice is reasonable in the imperative setting for which ALL(*) was developed, it renders the algorithm incomplete relative to a predicate-aware specification, because a prediction can send the parser down a path that leads to a predicate failure when a different path would have produced a successful parse. CoStar++ solves this problem by using a modified version of the ALL(*) prediction algorithm that evaluates predicates and actions only when doing so is necessary to guarantee completeness. CoStar++ semantic actions are pure functions, so speculatively executing them during prediction is safe.

This paper makes the following contributions:

- We present CoStar++, an extension of the CoStar verified ALL(*) parser interpreter that adds support for semantic predicates and actions. These new semantic features increase the expressivity of both the language definitions that the interpreter can accept and its output type.
- We present a modified version of ALL(*) prediction that CoStar++ uses to ensure completeness in the presence of semantic predicates.
- We prove that for all CFGs without left recursion, CoStar++ is sound, complete, and terminating with respect to a semantics-aware specification that takes predicates and actions into account.
- We prove that CoStar++ identifies uniquely correct semantic values, and that it detects syntactic ambiguity when semantic ambiguity is undecidable.
- We use CoStar++ to write grammars for four real-world data formats, three of which have non-context-free semantic specifications, and we show that CoStar++ achieves linear-time performance on benchmarks for these formats. As part of the evaluation, we integrate the tool with the Verbatim verified lexer interpreter [6,7] to create a fully verified front end for lexing and parsing data formats.

CoStar++ consists of roughly 6,500 lines of specification and 7,000 lines of proof. The grammars used in the performance evaluation comprise another 700 lines of specification and 100 lines of proof. CoStar++ and its accompanying performance evaluation framework are open source and available online [9].

```
Inductive json_value : Type :=
| JObj  (kv_pairs : list (string * json_value))
| JArr  (vs : list json_value)
| JBool (b : bool)
| JNum  (i : Z)
| JStr  (s : string)
| JNull.
```

Fig. 1: Algebraic data type representation of JSON values, shown in the concrete syntax of Gallina, the functional programming language embedded in Coq.

```
Value  ::= Object             ⟦λ(ps,_).nodup ps⟧?  ⟦λ(ps,_).JObj ps⟧!
         | Array                                   ⟦λ(vs,_).JArr vs⟧!
         | ...
Object ::= '{' Pair Pairs '}'                      ⟦λ(_,p,ps,_,_).p :: ps⟧!
         | '{'           '}'                       ⟦λ_.[]⟧!
...
```

Fig. 2: JSON grammar fragment annotated with semantic predicates and actions.

The paper is organized as follows. In §2, we introduce CoStar++ by example. We present the tool's correctness properties in §3. We then discuss the challenges of specifying the tool's behavior on ambiguous input (§4) and ensuring completeness after adding predicates to the tool's correctness specification (§5). In §6, we evaluate the tool's performance and describe the semantic features of the grammars used in the evaluation. Finally, we survey related work in §7.

2 CoStar++ by Example

In this section, we give an example of a simple grammar that includes a non-context-free semantic property, and we sketch the execution of the CoStar++ parser that this grammar specifies, with a focus on the parser's semantic features.

2.1 A Grammar for Parsing Duplicate-Free JSON

Suppose we want to use CoStar++ to define a JSON parser, and we only want the parser to accept JSON input in which objects contain no duplicate keys. The parser's output type might look like the algebraic data type (ADT) in Figure 1. To obtain a parser that produces values of this type, and that enforces the "unique keys" invariant, we can provide CoStar++ with the grammar excerpted in Figure 2. A CoStar++ grammar production has the form $X ::= \gamma\ [\![p]\!]?\ [\![f]\!]!$, where X is a nonterminal, γ is a sequence of terminals and nonterminals,[2] p is an optional semantic predicate, and f is a semantic action.

Semantic actions build the semantic values that the parser produces. An action is a function with a dependent type that is determined by the grammar symbols in the accompanying production. An action for production $X ::= \gamma$ has type $[\![\gamma]\!] \to [\![X]\!]$, where the semantic tuple type $[\![\gamma]\!]$ is computed as follows:

$$[\![\bullet]\!] = \mathbb{1}$$
$$[\![s\beta]\!] = [\![s]\!] \times [\![\beta]\!]$$

and $[\![s]\!]$ is a user-defined mapping from grammar symbols to semantic types. For the example grammar, $[\![\texttt{Value}]\!]$ = `json_value` (i.e., the parser produces a `json_value` each time it processes a `Value` nonterminal), and $[\![\texttt{Object}]\!]$ = `list (string * json_value)`.

In addition, productions are optionally annotated with semantic predicates. A predicate for production $X ::= \gamma$ has type $[\![\gamma]\!] \to \mathbb{B}$. At parse time, CoStar++ applies predicates to the semantic values that the actions produce and rejects the input when a predicate fails.

A production like this one:

$$\texttt{Value} ::= \texttt{Object} \quad [\![\lambda\texttt{(prs,\_).nodupKeys prs}]\!]? \quad [\![\lambda\texttt{(prs,\_).JObj prs}]\!]!$$

can be read as follows: "To produce a result of type $[\![\texttt{Value}]\!]$, first produce a tuple of type $[\![\texttt{Object}]\!]$ and apply predicate $[\![\lambda\texttt{(prs,\_).nodupKeys prs}]\!]?$ to it (where the `nodupKeys` function checks whether the string keys in an association list are unique). If the check succeeds, apply action $[\![\lambda\texttt{(prs,\_).JObj prs}]\!]!$ to the tuple."

2.2 Parsing Valid and Invalid Input

In Figure 3, we illustrate how CoStar++ realizes the example JSON grammar's semantics by applying CoStar++ to the grammar and tracing the resulting parser's execution on valid JSON input.

CoStar++ is implemented as a stack machine with a small-step semantics. At each point in its execution, the machine performs a single atomic update to its state based on its current configuration. Figure 3 shows the machine's stack at each point in the trace (other machine state components are omitted for ease of exposition). Each stack frame $[\alpha \ \& \ \bar{v}, \beta]$ holds a sequence of processed grammar symbols α, a semantic tuple $\bar{v} : [\![\alpha]\!]$ for the processed symbols, and a sequence of unprocessed symbols β. In the initial state σ_0, the stack consists of a single frame $[\bullet \ \& \ \texttt{tt}, \texttt{Value}]$ that holds an empty sequence of processed symbols $\bullet$, a semantic value of type $[\![\bullet]\!]$ (`tt`, the sole value of type `unit`), and a sequence of unprocessed symbols that contains only the start symbol `Value`.

[2] Throughout this paper, nonterminals begin with capital letters and terminals appear in single quotes. When it is necessary to distinguish between terminals and the literal values that they match, we write terminal names in angle brackets (e.g., `<int>` for a terminal that matches an integer).

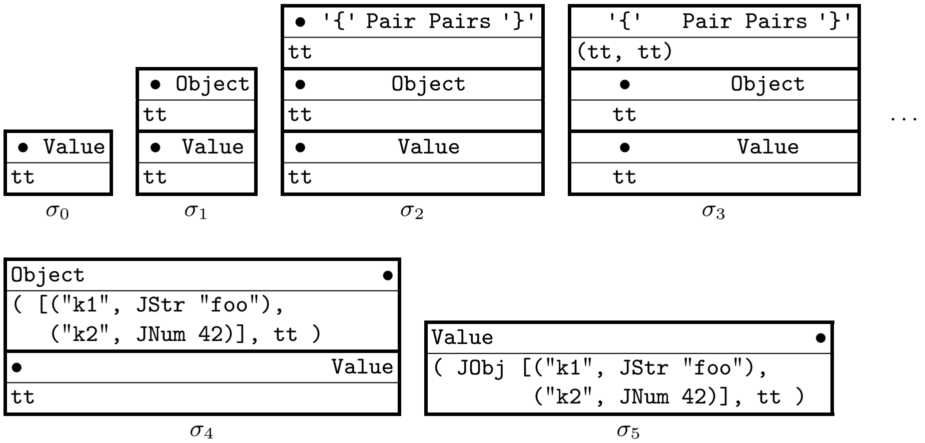

Fig. 3: Execution trace of a CoStar++ JSON parser applied to the valid string `{"k1": "foo", "k2": 42}`. A stack frame contains processed grammar symbols α (upper left portion of the frame), unprocessed grammar symbols β (upper right portion), and semantic tuple $\bar{v} : \llbracket \alpha \rrbracket$ (lower portion).

Each machine state also stores the sequence of remaining tokens. A token $(a \And v)$ is the dependent pair of a terminal symbol a and a literal value $v : \llbracket a \rrbracket$. (In our performance evaluation, we use a verified lexing tool that produces tokens of this type; see Section 6 for details.) In the Figure 3 example, the input string before tokenization is:

```
{"k1": "foo", "k2": 42}
```

Thus, in initial state σ_0, the machine holds tokens for the full input string:

(`'{'` & `tt`), (`<str>` & `"k1"`), (`':'` & `tt`), (`<str>` & `"foo"`) ...

In the transition from σ_0 to σ_1, the machine performs a **push** operation. A push occurs when the top stack symbol (the next unprocessed symbol in the top stack frame) is a nonterminal—`Value`, in this case. During a push, the machine examines the remaining tokens to determine which grammar right-hand side to push onto the stack. The prediction subroutine that performs this task is what distinguishes ALL(*) from other parsing algorithms. Parr et al. [14] describe the prediction mechanism in detail; in brief, the parser launches a subparser for each candidate right-hand side and advances the subparsers only as far as necessary to identify a uniquely viable choice. In the example, the prediction mechanism identifies the right-hand side `Object` as the uniquely viable choice and pushes it onto the stack in a new frame.

The transition from σ_1 to σ_2 is another push operation, in which the prediction mechanism identifies `'{' Pair Pairs '}'` as the unique right-hand side for nonterminal `Object` that may produce a successful parse. To transition from σ_2 to σ_3, the machine performs a **consume** operation. A consume occurs when the top stack symbol is a terminal a. The machine matches a against terminal a'

from the head remaining token. In this case, the top stack terminal '{' matches the terminal in token ('{' & tt), so the machine pops the token and stores its semantic value tt in the current frame.

After several more operations, the machine reaches state σ_4. At this point, the machine has fully processed nonterminal Object, producing a semantic value of type $[\![$Object$]\!]$ = list (string * json_value), there are no more symbols left to process in the top frame, and nonterminal Value in the frame below has not yet been fully processed (we call such a nonterminal "open", and the frame containing it the "caller" frame). In such a configuration, the machine performs a **return** operation, which involves the following steps:

1. The machine retrieves the predicate and action for the production being reduced. In the Figure 3 example, the production is Value ::= Object, the predicate is $[\![\lambda$(ps,_).nodup ps$]\!]$? (where the nodup function checks whether string keys in an association list are unique), and the action is $[\![\lambda$(ps,_).JObj ps$]\!]$!.
2. The machine applies the predicate to the semantic tuple $\bar{v}$ in the top frame. In the example, the predicate evaluates to true because the list of key/value pairs contains no duplicate keys.
3. If the predicate succeeds (as it does in the example), the machine applies the action to $\bar{v}$, producing a new semantic value v'. It then pops the top frame, moves the open nonterminal in the caller frame to the list of processed symbols, and stores v' in the caller frame. In this case, the machine makes Value a processed symbol (the nonterminal has now been fully reduced), and it stores v' = JObj [("k1", JStr "foo"), ("k2", JNum 42)] in the caller frame.

In state σ_5, the machine is in a final configuration; there are no unprocessed symbols in the top frame, and no caller frame to return to. In such a configuration, the machine halts and returns the semantic value it has accumulated for the start symbol. It tags the value as Unique or Ambig based on the value of another machine state component: a boolean flag indicating whether the machine detected ambiguity during the parse. In our example, the input is unambiguous, so the result of the parse is Unique (JObj [("k1", JStr "foo"), ("k2", JNum 42)]).

We now describe how the example JSON parser's behavior differs on the string {"k1": "foo", "k1": 42}, which is syntactically well-formed but violates the "no duplicate keys" property. During the first several steps involved in processing this string, the machine stacks match those in Figure 3. When the machine reaches a state that corresponds to state σ_4 in Figure 3, it attempts to perform a return operation by applying the predicate for production Value ::= Object to the list of key/value pairs [("k1", JStr "foo"), ("k1", JNum 42)]. This time, the predicate fails because of the duplicate keys, so the machine halts and returns a Reject value along with a message describing the failure.

3 Interpreter Correctness

In this section, we describe the CoStar++ interpreter's correctness specification and then present the interpreter's high-level correctness properties.

$$\boxed{\textsc{SemValueDer}: \; s \xrightarrow{v \,:\, \llbracket s \rrbracket} w}$$

$$\textsc{TerminalSemDer}\quad \frac{}{a \xrightarrow{v} (a \;\&\; v)}$$

$$\textsc{NonterminalSemDer}\quad \frac{X ::= \gamma\ \llbracket p \rrbracket?\ \llbracket f \rrbracket!\ \in \mathcal{G} \qquad \gamma \xrightarrow{\bar{v}} w \qquad p(\bar{v}) = \texttt{true}}{X \xrightarrow{f(\bar{v})} w}$$

$$\boxed{\textsc{SemValuesDer}: \; \gamma \xrightarrow{\bar{v} \,:\, \llbracket \gamma \rrbracket} w}$$

$$\textsc{NilSemDer}\quad \frac{}{\bullet \xrightarrow{\texttt{tt}} \epsilon}$$

$$\textsc{ConsSemDer}\quad \frac{s \xrightarrow{v} w_1 \qquad \beta \xrightarrow{\bar{v}} w_2}{s\beta \xrightarrow{(v,\bar{v})} w_1 w_2}$$

$$\boxed{\textsc{TreeDer}: \; s \xrightarrow{t \,:\, tree} w}$$

$$\textsc{TerminalLeafDer}\quad \frac{}{a \xrightarrow{\texttt{Leaf}(a)} (a \;\&\; v)}$$

$$\textsc{NonterminalNodeDer}\quad \frac{X ::= \gamma \in \mathcal{G} \qquad \gamma \xrightarrow{\bar{t}} w}{X \xrightarrow{\texttt{Node}(X,\bar{t})} w}$$

$$\boxed{\textsc{ForestDer}: \; \gamma \xrightarrow{\bar{t} \,:\, list\ tree} w}$$

$$\textsc{NilForestDer}\quad \frac{}{\bullet \xrightarrow{\bullet} \epsilon}$$

$$\textsc{ConsForestDer}\quad \frac{s \xrightarrow{t} w_1 \qquad \beta \xrightarrow{\bar{t}} w_2}{s\beta \xrightarrow{t,\bar{t}} w_1 w_2}$$

Fig. 4: Grammatical derivation relations for semantic values and parse trees.

3.1 Correctness Specification

CoStar++ is sound and complete relative to a grammatical derivation relation called SemValueDer with the judgment form $s \xrightarrow{v} w$, meaning that symbol s derives word w, producing semantic value v. Figure 4 shows this relation as well as a mutually inductive one, SemValuesDer, over sentential forms (grammar right-hand sides). This latter relation has the judgment form $\gamma \xrightarrow{\bar{v}} w$ (symbols γ derive word w, producing semantic tuple $\bar{v}$). In terms of predicates and actions, the key rule is NonterminalSemDer, which says that if (a) $X ::= \gamma\ \llbracket p \rrbracket?\ \llbracket f \rrbracket!$ is a grammar production; (b) the right-hand side γ derives word w, producing the semantic tuple $\bar{v}$; and (c) $\bar{v}$ satisfies predicate p, then applying action f to $\bar{v}$ produces a correct value for left-hand nonterminal X.

Portions of the correctness theorems refer to the existence of correct parse trees for the input. Parse tree correctness is defined in terms of a pair of mutually inductive relations, TreeDer and ForestDer (also in Figure 4). These relations are isomorphic to SemValueDer and SemValuesDer, but they produce parse trees and parse tree lists (respectively), where a parse tree is an n-ary tree with terminal-labeled leaves and nonterminal-labeled internal nodes.

3.2 Parser Correctness Theorems

The main CoStar++ correctness theorems describe the behavior of the interpreter's top-level `parse` function, which has the type signature shown in Figure

```
parse (g  : grammar)
      (Hw : grammar_wf g)
      (s  : nonterminal)
      (ts : list token) :
      parse_result s
```

(a) `parse` type signature

```
parse_result (x : nonterminal) :=
| Unique (v : ⟦x⟧)
| Ambig  (v : ⟦x⟧)
| Reject (s : string)
| Error  (e : parse_error)
```

(b) `parse` return type

Fig. 5: The type signature of the interpreter's top-level entry point (a), and the interpreter's return type (b).

5a. The `parse` function takes a grammar `g`, a proof that `g` is well-formed,[3] a start nonterminal `s`, and a token sequence `ts`. The function produces a `parse_result s`, a dependent type indexed by `s`. As shown in Figure 5b, a `parse_result x` is either a semantic value of type ⟦`x`⟧ tagged as `Unique` or `Ambig` (indicating whether the input is ambiguous), a `Reject` value with a message explaining why the input was rejected, or an `Error` value indicating that the stack machine reached an inconsistent state.

We list the CoStar++ high-level correctness theorems below, and we highlight several interesting aspects of their proofs in Sections 4 and 5. Each theorem assumes a non-left-recursive grammar $\mathcal{G}$.

Theorem 1 (Soundness, unique derivations). If `parse` applied to $\mathcal{G}$, nonterminal S, and word w returns a semantic value `Unique`(v), then v is the sole correct semantic value for S and w.

Theorem 2 (Soundness, ambiguous derivations). If `parse` applied to $\mathcal{G}$, nonterminal S, and word w returns a semantic value `Ambig`(v), then v is a correct semantic value for S and w, and there exist two correct parse trees t and t' for S and w, where $t \neq t'$.

Theorem 3 (Error-free termination). The interpreter never returns an `Error` value.

Theorem 4 (Completeness). If v is a correct semantic value for nonterminal S and word w, then either (a) v is the sole correct semantic value for S and w and the interpreter returns `Unique`(v), or (b) multiple correct parse trees exist for S and w, and the interpreter returns a correct semantic value `Ambig`(v').

The theorems above have been mechanized in Coq. Each theorem has a proof based on (a) an invariant I over the machine state that implies the high-level theorem when it holds for the machine's final configuration; and (b) a preservation lemma showing that each machine operation (push, consume, and return) preserves I. Section 5.2 contains an example of such an invariant.

[3] Internally, a CoStar++ grammar is a finite map in which each base production $X ::= \gamma$ maps to an annotated production $X' ::= \gamma'\ [\![p]\!]?\ [\![f]\!]!$. The well-formedness property says that $X = X'$ and $\gamma = \gamma'$ for each key/value pair in the map. This property enables the interpreter to retrieve the predicate and action for key $X := \gamma$.

```
X ::= <int> Y          ⟦λ(i,(s,_),_).i - String.length s⟧!
    | Z <bool>         ⟦λ((_,s),b,_).if b then String.length s else 0⟧!
Y ::= <string> <bool>  ⟦λ(s,b,_).(s,b)⟧!
Z ::= <int> <string>   ⟦λ(i,s,_).(i,s)⟧!
```

Fig. 6: Grammar that recognizes an `<int><string><bool>` sequence. For some inputs, two different syntactic derivations produce the same semantic value.

4 Semantic Actions and Ambiguity

There is an apparent type mismatch between the "unique" and "ambiguous" soundness theorems in Section 3. According to Theorem 1, a `Unique`(v) parse result indicates that v is a uniquely correct *semantic value* for the input, while Theorem 2 says that an `Ambig`(v) result implies the existence of multiple correct *parse trees* for the input. The reason for this asymmetry is that syntactically ambiguous inputs may not be ambiguous at the semantic level; actions can map two distinct parse trees for an input to the same semantic value, and predicates can eliminate semantic ambiguity by rejecting semantic values as malformed. For these reasons, the problem of identifying semantic ambiguity is undecidable when semantic values lack decidable equality. When CoStar++ flags an ambiguous input, it is only able to guarantee that ambiguity exists at the syntactic level.

We illustrate this point with an example involving the somewhat contrived grammar in Figure 6. Start symbol `X` matches an `<int><string><bool>` sequence in two possible ways—one involving the first right-hand side for `X`, and one involving the second right-hand side. These two right-hand sides can be used to derive two distinct parse trees for such a token sequence (we represent leaves as terminal symbols for readability):

(1a) `Node X [<int>, Node Y [<string>, <bool>]]`
(1b) `Node X [Node Z [<int>, <string>], <bool>]`

However, while any `<int><string><bool>` sequence is ambiguous at the syntactic level, only some inputs are semantically ambiguous. For example, on input

(`<int>` & `10`) (`<string>` & `"apple"`) (`<bool>` & `false`)

the actions attached to the two right-hand sides for `X` produce two distinct values:

(2a) `10 - String.length "apple" = 5`
(2b) `if false then String.length "apple" else 0 = 0`

However, replacing the literal value in the `<bool>` token with `true` makes the two derivations produce the same semantic value:

(3a) `10 - String.length "apple" = 5`
(3b) `if true then String.length "apple" else 0 = 5`

In theory, when CoStar++ identifies multiple semantic values for these examples, it could determine whether the input is semantically ambiguous by comparing the values, because integer equality is decidable. However, semantic types are user-defined, and we do not require them to have decidable equality; the user may want the interpreter to produce functions or other incomparable values. Therefore, in the general case, the interpreter can only certify that the input has two distinct parse trees—this guarantee is the one that Theorem 2 provides.

5 Semantic Predicates and Completeness

One of the main challenges of implementing and verifying CoStar++ was ensuring completeness in the presence of semantic predicates. ALL(*) is a predictive parsing algorithm; at decision points, it launches subparsers that speculatively explore alternative paths. ALL(*) as originally described [14] does not apply semantic actions or check CoStar++-style predicates at prediction time. However, a predicate-oblivious prediction algorithm results in an interpreter that is incomplete relative to the SemValueDer specification (Figure 4). In other words, it can make a choice that eventually causes the interpreter to reject input as invalid due to a failed predicate, when a different choice would have led to a successful parse. In this section, we present a modification to the ALL(*) prediction mechanism and prove that it makes the interpreter complete with respect to its semantic specification.

5.1 A Semantics-Aware Prediction Mechanism

The semantics-aware version of CoStar++ uses a modified version of ALL(*) prediction that is guaranteed not to send the interpreter down a “bad path.” In designing this modification, we faced a tradeoff between speed and expressiveness; checking predicates and building semantic values along all prediction paths is expensive, but it is sometimes necessary to ensure completeness.

Our solution leverages the fact that the original ALL(*) prediction mechanism addresses a similar problem; it is actually a combination of two prediction strategies that make different tradeoffs with respect to speed and expressiveness:

- **SLL prediction** is an optimized algorithm that ignores the initial parser stack at the start of prediction. As a result, subparser states are compact and recur frequently, which makes them amenable to caching. The tradeoff is that because of the missing context, SLL prediction must sometimes overapproximate the parser’s behavior by simulating a return to *all* possible contexts.
- **LL prediction** is a slower but sound algorithm in which subparsers have access to the initial parser stack; the algorithm is thus a precise nondeterministic simulation of the parser’s behavior. When the SLL algorithm detects an ambiguity, the prediction mechanism fails over to the LL strategy to determine whether the ambiguity is genuine or involves a spurious path introduced by the overapproximation; using the result of SLL prediction directly in such a case would render the parser incomplete.

Semantics-aware prediction works as follows:

- SLL prediction is unchanged; subparsers do not build semantic values or check semantic properties. SLL is thus still an overapproximation of the parser; not evaluating the predicates is equivalent to assuming that they succeed.
- LL prediction builds semantic values and checks semantic properties along all paths. It thus remains a precise nondeterministic simulation of the parser.

This approach assumes that most predictions are unambiguous without considering predicates, and the more expensive LL strategy is thus rarely required.

5.2 A Backward-Looking Completeness Invariant

Adding semantic features to LL prediction makes CoStar++ complete with respect to the SemValueDer specification. Theorem 4 (the interpreter completeness theorem) relies on the following lemma:

Lemma 1 (Completeness modulo ambiguity detection). If v is a correct semantic value for nonterminal S and word w, then there exists a semantic value v' such that the interpreter returns either `Unique`(v') or `Ambig`(v') for S and w.

In essence, this lemma says that the interpreter does not reject valid input. Its proof is based on an invariant over the machine state guaranteeing that no machine operation can result in a rejection.

In the absence of semantic predicates, a natural definition of this invariant says that the concatenated unprocessed stack symbols recognize the remaining token sequence. Such an invariant is purely forward-looking; it refers only to symbols and tokens that the interpreter has not processed yet. However, this invariant is too weak to prove that CoStar++ never rejects valid input, because a predicate can fail on semantic values that were produced by earlier machine steps. To rule out such cases, we need an invariant that is both backward- and forward-looking; i.e., one that refers to both the "past" and "future" of the parse.

The CoStar++ completeness invariant, StackAcceptsSuffix_I, appears in Figure 7. It holds when the remaining tokens can be split into a prefix w_1 and suffix w_2 such that the unprocessed symbols β in the top stack frame produce a semantic tuple for w_1, and the auxiliary invariant FramesAcceptSuffix_I holds for the lower frames and w_2.

The FramesAcceptSuffix_I definition (also in Figure 7) is parametric over symbols γ and semantic tuple $\bar{v} : [\![\gamma]\!]$. The $\bar{v}$ parameter represents the "incoming" tuple during the eventual return operation from the frame above the ones in scope. The base case of FramesAcceptSuffix_I says that if the list of remaining frames is empty, then the remaining token sequence must be empty as well. In the case of a non-empty list of frames, the following properties hold:

- The remaining tokens can be split into a prefix w_1 and suffix w_2 such that the unprocessed symbols in the head frame produce a semantic tuple for w_1. This property (which appears in StackAcceptsSuffix_I as well) is the forward-looking portion of the invariant.

$$\boxed{\textsc{FramesAcceptSuffix\_I} : (\bar{v} : [\![\gamma]\!]), \phi \ \triangleright\ w}$$

$$\textsc{FramesAcceptSuffix\_Nil}\qquad \frac{}{\bar{v}, \bullet \ \triangleright\ \epsilon}$$

$$\textsc{FramesAcceptSuffix\_Cons}\qquad \frac{\begin{array}{c} \bar{v}_\gamma : [\![\gamma]\!] \qquad \bar{v}_\alpha : [\![\alpha]\!] \qquad \bar{v}_\beta : [\![\beta]\!] \qquad p : [\![\gamma]\!] \to \mathbb{B} \qquad f : [\![\gamma]\!] \to [\![X]\!] \\ \beta \xrightarrow{\bar{v}_\beta} w_1 \qquad X ::= \gamma\ [\![p]\!]?\ [\![f]\!]! \in \mathcal{G} \qquad p(\bar{v}_\gamma) = \texttt{true} \\ \texttt{revTup(}\bar{v}_\alpha\texttt{)}\ \ [\![+\!\!+]\!]\ \ (f(\bar{v}_\gamma), \bar{v}_\beta), \phi \ \triangleright\ w_2 \end{array}}{\bar{v}_\gamma, [\alpha \ \&\ \bar{v}_\alpha, X\beta]\phi \ \triangleright\ w_1 w_2}$$

$$\boxed{\textsc{StackAcceptsSuffix\_I} : \phi \ \blacktriangleright\ w}$$

$$\frac{\bar{v}_\alpha : [\![\alpha]\!] \qquad \bar{v}_\beta : [\![\beta]\!] \qquad \beta \xrightarrow{\bar{v}_\beta} w_1 \qquad \texttt{revTup(}\bar{v}_\alpha\texttt{)}\ \ [\![+\!\!+]\!]\ \bar{v}_\beta, \phi \ \triangleright\ w_2}{[\alpha \ \&\ \bar{v}_\alpha, \beta]\phi \ \blacktriangleright\ w_1 w_2}$$

Fig. 7: The StackAcceptsSuffix_I machine state invariant over stack ϕ and token sequence w. The invariant guarantees that the interpreter does not reject valid input. The $[\![+\!\!+]\!]$ function concatenates two semantic tuples, and the `revTup` function reverses a semantic tuple.

- There exists a grammar production $X ::= \gamma\ [\![p]\!]?\ [\![f]\!]!$, where X is the open nonterminal in the head frame and γ is the right-hand side from the frame above, such that semantic tuple $\bar{v}_\gamma$ from the frame above satisfies p. This condition is the backward-looking portion of the invariant.
- FramesAcceptSuffix_I holds for the remaining frames and w_2.

Lemma 2 (Completeness invariant prevents rejection). If StackAcceptsSuffix_I holds at machine state σ, then a machine transition out of σ never produces a `Reject` result.

Lemma 3 (Preservation of completeness invariant). If StackAcceptsSuffix_I holds at machine state σ and $\sigma \rightsquigarrow \sigma'$, then StackAcceptsSuffix_I holds at state σ'.

6 Performance Evaluation

We evaluate CoStar++'s parsing speed and asymptotic behavior by extracting the tool to OCaml source code and recording its execution time on benchmarks for four real-world data formats. In each experiment, we provide CoStar++ with a grammar for a data format to obtain a parser for that format, and we record the parser's execution time on valid inputs of varying size. The benchmarks are as follows:

- **JSON** is a popular format for storing and exchanging structured data. The actions in our JSON grammar build an ADT representation of a JSON value with a type similar to the one in Figure 1. The predicates ensure that JSON objects contain no duplicate keys. The JSON data set contains biographical information for US Members of Congress [1].
- **PPM** is a text-based image file format in which each pixel is represented by a triple of (red, green, blue) values. A PPM file includes a header with numeric values that specify the image's width and height, and the maximum value of any pixel component. The actions in our PPM grammar build a record that contains the header values and a list of pixels. The predicates validate the non-context-free dependencies between the image's header and pixels. We generated a PPM data set by using the ImageMagick command-line tool `convert` to convert a single PPM image to a range of different sizes.
- **Newick trees** are an ad hoc format for representing arbitrarily branching trees with labeled edges. They are used in the evolutionary biology community to represent phylogenetic relationships. The Newick grammar's actions convert an input to an ADT representation of an arbitrarily branching tree. Our Newick data set comes from the 10kTrees Website, Version 3 [2], a public database of phylogenetic trees for various mammalian orders.
- **XML** is a widely used format for storing and transmitting structured data. An XML document is a tree of elements; each element begins and ends with a string-labeled tag, and the labels in corresponding start and end tags must match—a non-context-free property in the general case where the set of valid labels is infinite. The actions in our XML grammar build an ADT representation of an XML document, and the predicates check that corresponding tags contain matching labels. Our XML data set is a portion of the Open American National Corpus [13], a collection of English texts with linguistic annotations.

CoStar++ requires tokenized input. We use the Verbatim verified lexer interpreter [6,7] to obtain lexers for all four formats. In the benchmarks, we use these lexers to pre-tokenize each input before parsing it.

We ran the CoStar++ benchmarks on a laptop with 4 2.5 GHz cores, 7 GB of RAM, and the Ubuntu 16.04 OS. We compiled the extracted CoStar++ code with OCaml compiler version 4.11.1+flambda at optimization level -O3.

The CoStar++ benchmark results appear in Figure 8. Each scatter plot point represents the parse time for one input file, averaged over ten trials. While the worst-case time complexity of ALL(*) is $O(n^4)$ [14], and CoStar++ lacks an optimization based on the *graph-structured stack* data structure [16] that factors into this bound, the tool appears to perform linearly on the benchmarks. For each set of results, we compute a least-squares regression line and a Locally Weighted Scatterplot Smoothing (LOWESS) curve [3]. LOWESS is a non-parametric technique for fitting a smooth curve to a set of data points; i.e., it does not assume that the data fit a particular distribution, linear or otherwise. The LOWESS curve and regression line correspond closely for each set of results, suggesting that the relationship between input size and execution time is linear.

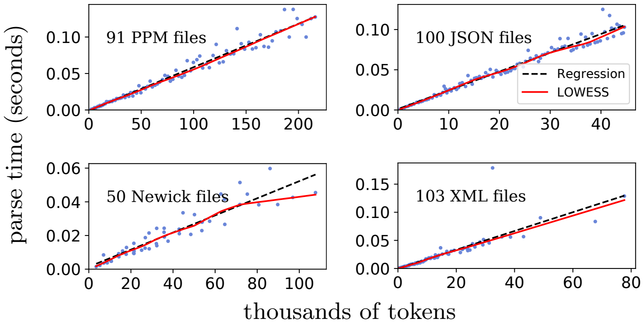

Fig. 8: Input size vs. CoStar++ average execution time on four benchmarks.

7 Related Work

CoStar++ builds on CoStar [11], another tool based on the ALL(*) algorithm and verified in Coq. CoStar produces parse trees that are generic across grammars modulo grammar symbol names. It is correct in terms of a specification in which a parse tree is the witness to a successful derivation. CoStar++ improves upon this work by supporting semantic actions and predicates.

ALL(*) was developed for the ANTLR parser generator [14]. While ALL(*) as originally described and as implemented in ANTLR supports a notion of semantic predicates, its prediction mechanism does not execute semantic actions, and thus cannot evaluate predicates over the results of those actions. The original algorithm is therefore incomplete with respect to our predicate-aware specification. These design choices are reasonable in terms of efficiency, and in terms of correctness in an imperative setting. It is potentially expensive to execute predicates and actions along a prediction path that the parser does not ultimately take. More importantly, doing so can produce counterintuitive behavior when the actions alter mutable state in ways that cannot be easily undone. These concerns do not apply to our setting, in which semantic actions are pure functions.

Several existing verified parsers for CFGs support some form of semantic actions. Jourdan et al. [8] and Lasser et al. [10] present verified parsing tools based on the LR(1) and LL(1) parsing algorithms, respectively. Both tools represent a semantic action as a function with a dependent type computed from the grammar symbols in its associated production. CoStar++ uses a similar representation of predicates and actions. Edelmann et al. [5] describe a parser combinator library and an accompanying type system that ensures that any well-typed parser built from the combinators is LL(1); such a parser therefore runs in linear time. Danielsson [4] and Ridge [15] present similar parser combinator libraries that can represent arbitrary CFGs but do not provide the linear runtime guarantees of LL(1) parsing.

Acknowledgments Sam Lasser's research was supported by a Draper Scholarship.

References

1. congress-legislators database (2022), https://github.com/unitedstates/congress-legislators
2. Arnold, C., Matthews, L.J., Nunn, C.L.: The 10kTrees Website: A New Online Resource for Primate Phylogeny. Evolutionary Anthropology: Issues, News, and Reviews **19**(3), 114–118 (2010)
3. Cleveland, W.S.: Robust Locally Weighted Regression and Smoothing Scatterplots. Journal of the American Statistical Association **74**(368), 829–836 (1979)
4. Danielsson, N.A.: Total Parser Combinators. In: International Conference on Functional Programming (2010), https://doi.org/10.1145/1863543.1863585
5. Edelmann, R., Hamza, J., Kunčak, V.: Zippy LL(1) Parsing with Derivatives. In: Programming Language Design and Implementation (2020), https://doi.org/10.1145/3385412.3385992
6. Egolf, D., Lasser, S., Fisher, K.: Verbatim: A Verified Lexer Generator. In: LangSec Workshop (2021), https://langsec.org/spw21/papers.html#verbatim
7. Egolf, D., Lasser, S., Fisher, K.: Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Derivatives. In: Certified Programs and Proofs (2022), https://doi.org/10.1145/3497775.3503694
8. Jourdan, J., Pottier, F., Leroy, X.: Validating LR(1) Parsers. In: European Symposium on Programming (2012), https://doi.org/10.1007/978-3-642-28869-2_20
9. Lasser, S., Casinghino, C., Egolf, D., Fisher, K., Roux, C.: GitHub repository for the CoStar++ development and performance evaluation framework (2022), https://github.com/slasser/CoStar
10. Lasser, S., Casinghino, C., Fisher, K., Roux, C.: A Verified LL(1) Parser Generator. In: Interactive Theorem Proving (2019), https://doi.org/10.4230/LIPIcs.ITP.2019.24
11. Lasser, S., Casinghino, C., Fisher, K., Roux, C.: CoStar: A Verified ALL(*) Parser. In: Programming Language Design and Implementation (2021), https://doi.org/10.1145/3453483.3454053
12. Momot, F., Bratus, S., Hallberg, S.M., Patterson, M.L.: The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In: IEEE Cybersecurity Development (2016), https://doi.org/10.1109/SecDev.2016.019
13. Open American National Corpus (2010), http://www.anc.org/data/oanc/download/
14. Parr, T., Harwell, S., Fisher, K.: Adaptive LL(*) Parsing: The Power of Dynamic Analysis. Object-Oriented Programming, Systems, Languages, and Applications (2014), https://doi.org/10.1145/2660193.2660202
15. Ridge, T.: Simple, Functional, Sound and Complete Parsing for All Context-Free Grammars. In: Certified Programs and Proofs (2011), https://doi.org/10.1007/978-3-642-25379-9_10
16. Scott, E., Johnstone, A.: GLL Parsing. Electronic Notes in Theoretical Computer Science **253**(7), 177–189 (2010), https://doi.org/10.1016/j.entcs.2010.08.041