Formal Development of a Top-Down Chart Parser for Parsing Expression Grammars

Zephyr S. Lucas[1] and Natarajan Shankar[2]

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park CA 94025 USA
shankar@csl.sri.com

Abstract. Formalization is often seen as an exclamation point certifying the correctness of a mathematical development. However, in the context of an interactive proof assistant, formalization is the medium for a creative dialogue that identifies gaps, tags errors, and refines the definitions, claims, and proofs to enhance the correctness, clarity, elegance, robustness, and reusability of mathematical content. With this goal in mind, we explore the formalization of a top-down chart parser for parsing expression grammars (PEGs) using SRI's Prototype Verification System (PVS). The parser, which supports dynamic loop detection, operates on a chart-recording data structure that also embeds the parsing stack. Since the parsing algorithm is tail-recursive and operates on fixed storage, it is hardware-friendly. The invariants needed to establish the correctness of the chart parser are complex and delicate. They were derived through an iterative proof exploration process. The generated proof obligations and proof attempts led to a number of significant corrections and improvements to the formal development.

Keywords: PVS, PEG grammar, chart parsing, verified parser, top-down parsing

1 Introduction

Parsing is a fundamental operation for transforming concrete representations of data such as documents and program text into their abstract machine-representable counterparts [**?**]. Even though parsing is one of the most developed fields of computing, a number of significant computing vulnerabilities have been attributable to parsing errors. For example, parsing errors were detected in the unverified front-end of the verified CompCert compiler [**?**], and security vulnerabilities [**?**] such as Heartbleed [**?**] and SIGRed [**?**] stem from improper input validation attributable to parsing errors. Such shortcomings are especially lamentable given that language theory and parsing are amenable to rigorous formalization. We present the formalization of a top-down chart parsing interpreter for PEG grammars developed using SRI's Prototype Verification System (PVS) [**?**]. Equal in

importance to the correctness guarantees derived from the formalization is the iterative proof development process by which formalization is achieved. We describe how through an interactive dialogue, a proof assistant accelerates the construction of a correct, robust, and reusable formalization.

Parsers [**?**] can operate

- *Top-down*, by consuming input token strings matching a given grammar category, e.g., recursive descent and $LL(k)$ parsers, and Earley parsing for arbitrary context-free grammars, or
- *Bottom-up*, where the contiguous tokens are grouped into grammar categories, e.g., precedence parsers, LR(k) (and variants SLR, LALR, CLR), as well as GLR and CYK parsers for arbitrary context-free grammars.

Parsing algorithms like the Earley and CYK algorithms are examples of *chart parsers* [**?**,**?**,**?**] based on dynamic programming where the goal is to progressively label larger and larger spans of the input string with terminals and nonterminals based on the labels of contiguous subspans. Earley parsing takes a top-down approach whereas the CYK and Valiant algorithms take a bottom-up approach to parsing.

Our chart parser employs a table or a *scaffold* mapping each position in the input string and non-terminal to an entry that represents the parse result for the non-terminal at the given position. Our chart parser is a tail-recursive state machine that operates on fixed storage so that it can be easily mapped to a hardware implementation. With respect to formalization, the invariants associated with the parse table are complex since the table encapsulates a web of relationships that are impacted by an update to even a single entry. Using PVS as an interactive proof assistant greatly assisted in managing this complexity. More strongly, it would be extremely hard to construct or understand such proofs without the aid of a proof assistant.

In the present work, we focus on a class of grammars called Parsing Expression Grammars (PEGs) introduced by Ford [**?**]. These grammars are unambiguous and efficient for both top-down and bottom-up parsing. PEGs replace the unordered choice operation $e_1 \mid e_2$ in CFGs with an ordered choice operation e_1/e_2. In addition to concatenation $e_1 e_2$, the grammar supports an check operation $\&e$ that looks ahead for a match to e without consuming any tokens, and negation $!e$ that succeeds consuming no tokens only when a match against e fails. PEG grammars are a good match for recursive descent parsers. Parser combinators can be defined for each PEG construct that combine parsers for the grammar sub-expressions into a parser for the expression. PEG grammars can be *ill-formed* and contain loops, for example in the case of left recursion where, for example, a nonterminal n is mapped to the expression n/n'. PEG grammars can be checked for *well-formedness* through a static analysis, where well-formed grammars always yield parses that terminate in success or failure, however such an analysis can miss cases where the grammar is in fact loop-free.

Prior research has explored the formalization of PEG parsing where the grammars are conservatively analyzed for termination using recursive descent parsers [**?**,**?**]. Top-down parsers that have been verified include the CakeML

parser [**?**], an LL(1) parser [**?**], an SLR parser generator [**?**], and a recursive descent parser for context-free languages [**?**]. We believe that our work is the first to address the verification of an Earley-style chart parsing algorithm. We allow ill-formed grammars but extend the parser to dynamically check for loops. In this more general setting, it is possible to demonstrate that the parser is well-behaved on well-formed grammars while still admitting potentially ill-formed grammars.

We also formalize the parser interpreter as a state machine by internalizing the stack. The parser is centered around a table A where for each position i on the input string, there is an array $A(i)$ such that for each nonterminal n, $A(i)(n)$ is a table entry. Initially, all the table entries $A(i)(n)$ are *pending* indicating that parsing has not yet been initiated at any string position i for any nonterminal n. Once the root query has been launched for non-terminal r at position 0, we place a pair representing $(0, r)$ on the stack. The production $r \leftarrow e$ corresponding to the nonterminal r is evaluated, and parsing is continued through a case analysis on e. This can lead to pushing more position/nonterminal pairs on to the stack or to *finalized* entries representing failed, loopy, or successful parses. When a parse query is initiated on a position/nonterminal pair that is already on the stack, a loop is signaled, and this loop is propagated back through the stack to the root query.

We arrived at the above algorithm in a series of steps. We first started by formalizing the *scaffolding automata* algorithm due to Loff, Moreira, and Reis [**?**]. This is a bottom-up, right-to-left algorithm that fills in the scaffold from the final position backwards. In each iteration, it fills out the entries for a given position for each nonterminal. It is possible to sort the nonterminals so that the sub-queries for each nonterminal at a given position have already been processed. This variant would be quite straightforward to verify since the sequencing of the operations ensures that each entry is consistent with the entries corresponding to the sub-queries. Since it is challenging to write a grammar that passes a termination check, we allowed the nonterminals to be unordered. This introduces the possibility of loops. For example, a grammar with $S \leftarrow RS$, where R happens to succeed without consuming any tokens. We tried an alternative that involved using a stack to process any as yet unprocessed sub-queries. The use of the stack complicated the picture. We decided that if we were going to use a stack, then we might as well employ a top-down algorithm. We then formalized a top-down algorithm with an explicit stack argument, and soon noticed that the stack could be embedded within the scaffold itself. We then wrote out a full state machine and a few days of dialogue with the proof assistant yielded the invariants for the parsing algorithm. We then noticed that the flat state machine could be refactored to make the proofs less repetitive and more compact. This required additional strengthenings of the invariants suggested by a couple more days of dialogue with the proof assistant. Many of the definitions and invariants would have been painfully difficult to derive without the benefit of the interactive dialogue supported by powerful automation. Many of the proofs employ a heavy amount of automation even when it is obvious from experience

that the automation will succeed. Others need to be handled more delicately since they might fail and we as users need to understand the root cause of the failure.

The parsing algorithm we formalize is attractive since it captures top-down PEG parsing as a state machine with a single statically allocated data structure. This makes it hardware-friendly. We added dynamic loop detection in order to support a more liberal class of PEGs that admit cyclic dependencies and to address a novel verification challenge. If the grammar does not contain cyclic dependencies, the loop detection can be turned off. Our basic algorithm can also support certain optimizations for speeding up parsing. The chart parsing algorithm presents several challenges for verification. The key result we want at the end is that the parsing operation yields a consistent scaffold yielding a chart labeling the root query. Defining a consistent scaffold is itself quite challenging. The conditions on partially completed scaffold have to be characterized in such a way that each step of the parsing algorithm preserves this condition as an invariant, and the resulting completed chart is consistent. The final challenge is structuring the proof in order to make it easy to demonstrate that the invariant is preserved.

2 A Top-Down Chart Parser for PEGs

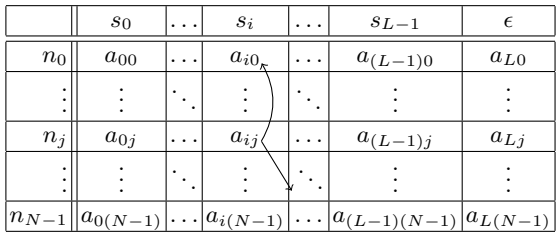
We employ a Chomsky Normal Form representation of PEG grammars where the right hand sides of productions are flat, i.e., contain no nesting of PEG operations. By definition, any PEG grammar can easily be represented in Chomsky Normal Form, whereas the corresponding transformation for CFGs is more complicated. PEG normal form requires each production to be of the form $n_j \leftarrow e$, where n_j is a non-terminal, and e is an expression of one of the eight types shown in Table 1.

Empty	ϵ
Failure	f
Any	$any(p)$
Terminal	c
Concatenation	$n_j n_k$
Ordered Choice	n_j / n_k
Check	$\&n_j$
Negation	$!n_j$

Table 1: Each row in contains one of the set of fundamental PEG normal-form operations we implemented. Here n_j, n_k are non-terminals, c is a character in our character space, and p is a predicate over our character space.

A generalized version of the scaffold data structure is shown in Figure 1.

As seen in Figure 1, the rows of the scaffold are determined by the non-terminals in the normal form of the grammar describing the language to be



	s_0	...	s_i	...	s_{L-1}	ϵ
n_0	a_{00}	...	a_{i0}	...	$a_{(L-1)0}$	a_{L0}
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
n_j	a_{0j}	...	a_{ij}	...	$a_{(L-1)j}$	a_{Lj}
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
n_{N-1}	$a_{0(N-1)}$	...	$a_{i(N-1)}$	...	$a_{(L-1)(N-1)}$	$a_{L(N-1)}$

Fig. 1: A generic scaffold for a PEG consisting of N productions (the rows) over an input token string s of length L (the columns)

parsed. The columns of the scaffold are determined by each character in the input string in left-to-right order. A given PEG grammar with normal-form productions (i.e. non-terminals) of cardinality N and input string s of length L determines a statically-sized scaffold of size N by $(L + 1)$. The size is $L + 1$ to account for an implicit empty ϵ that terminates the input string. As described in Section 1, our chart-parsing algorithm initializes each scaffold entry a_{in} with the value `pending`. The algorithm starts with entry a_{00}, and proceeds in a recursively top-down manner, updating each entry a_{in}.

Central to our parser is a table data structure, the scaffold, which is similar to the tables used by Earley and CYK parsers. The scaffold is a two-dimensional array containing entries a_{in} where i is a position in the input string s with $0 \leq i \leq |s|$, and n is a nonterminal from the given grammar. Each entry a_{in} will have the value `pending` (represented by a blank in Figures 2 to 4, `fail(height)` (represented by f), `loop` (represented by l), `good(height, span)` (represented by $g_{\texttt{span}}$, or `push(position, nonterminal)` (abbreviated as p_{in}). The `push` value represents the recursive application of that production, thus embedding the representation of the recursion stack into the statically-sized scaffold data structure itself; a more hardware friendly design. Loops are detected when the algorithm attempts a `push` for a scaffold entry that already contains a `push`; intuitively this indicates the algorithm has returned to a previously visited position in the scaffold and is about to push again, a push that will result in returning to the entry yet again, ad infinitum. Our chart-parsing algorithm initializes each scaffold entry a_{in} with the value `pending`. The algorithm starts with entry a_{00}, and proceeds in a recursively top-down manner, updating each entry a_{in}.

2.1 Example: Matching Parentheses

Consider the language of matching parentheses where we want the parser to accept the longest prefix of the input where parentheses have been matched A simple PEG for this can be written as

$$n_{start} \leftarrow \text{"(" } n_{start} \text{ ")" } n_{start} / \epsilon,$$

however this is not in normal form. We can rewrite it into a normal form with seven productions, each of which is exactly one of the forms shown in Table 1.

These productions are listed in the leftmost column of the scaffold in Figure 2. Because, once in normal form, each non-terminal corresponds to exactly one production, these will also be the rows of the scaffold. In Figure 2, we compute matching parenthesis on the input "`(()())()`".

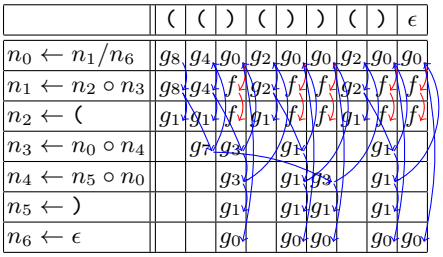


Fig. 2: Matching Parenthesis on "`(()())()`". This shows the contents of the scaffold after the parsing process has completed. Blue arrows show the successful parse tree, and red arrows show the proof of a failed parse. Scaffold entries marked g_i indicate an entry of *good* with a span of i, and entries marked f indicate a fail.

2.2 Example: Power of Two

If we now want to accept any input stream that has is a positive power of two length or is empty (the length of the input is 0 or one of $\{2, 4, 8, 16, \ldots\}$). The grammar

$$n_{start} \leftarrow n_{pow}!\,. \qquad n_{pow} \leftarrow\ .\,n_{pow}\,./\,.\,(\&n_{pow})\,./\epsilon$$

where "." is any character, specifies this language. Figure 3 shows this grammar running on the input "`parsed`".

2.3 Example: Loop

We can also consider a grammar which is not well formed. The following will accept strings that are a sequence of *blocks* ending with a `#`. Each block is either `a`, `b`, or `ab` (but not `ba`, which would instead be parsed as two separate blocks). The grammar

$$n_{start} \leftarrow n_{block}(\texttt{\#}/n_{start}) \qquad n_{block} \leftarrow !\texttt{c}(\texttt{a}/\epsilon)(\texttt{b}/\epsilon)$$

correctly parses all correct instances of this. Further it will successfully reject any string that contains the character `c`. This means on the language $\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{\#}\}$ it will behave properly, however if it encounters something other than `a`, `b`, `c`, or `#` it will loop. Figure 4 shows this grammar mid-computation on the input `abbda#`.

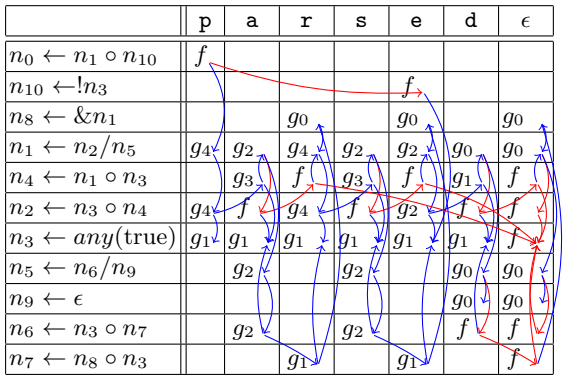


Fig. 3: Power of two length on "`parsed`". The rows have been rearranged to make it easier to follow the arrows. Note in the n_1 row there reason to believe this grammar would accept inputs of length 2, 4, and 0. This is because the n_{10} just checks that n_1 consumed all the input, so if the string started with the `r` (meaning it had length 4), then n_0 would correctly accept and consume all 4 characters.

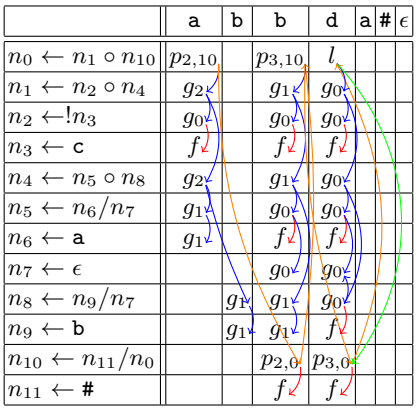


Fig. 4: Computation of the block grammar on the string "`abbda#`". The scaffold is shown midway through its computation at the time it detected the loop. The entries in the form $p_{i,j}$ show `push` entries with position i and non-terminal j, and the l entry shows a `loop`. The orange arrows between the `push` entries is the embedded call stack, while the green arrow shows the relationship that first detected the loop. If this scaffold were to continue running, the `push` entries would become `loop` entries as the program backed its way up the embedded stack where it will terminate and declare a loop. Note if this `d`, were a `c` then it would have already rejected by this point, and if the `d` were an `a` or a `b`, then it would eventually accept the whole input.

3 Chart Parsing: Formalization and Proof

We now present a formalization (*available on request*) of the chart parsing algorithm in PVS. This formalization exploits specific features of the PVS depen-

dently subtyped higher-order logic. We quickly summarize the basic features and explain more advanced features of the PVS specification language as needed to understand the formalization.

The PVS specification language is based on strongly typed higher-order logic. A PVS specification is a collection of theories. Each theory has some type and individual parameters, and the body of the theory is a sequence of declarations of types and constants. A type can be one of the built-in types; a (possibly dependent) tuple, record, or function type, a predicate subtype of the form $\{x : T \mid p(x)\}$, or an algebraic or coalgebraic datatype. The type `nat` of natural numbers is a predicate subtype of the type `int` of integers, which in turn is a subtype `rat` of the rational numbers, and the real number type `real`. Dependent types can be used, for example, to define the type of permutations over a given initial segment of the natural numbers as a dependent record with a `size` field over `nat`, and a permutation field that is an bijective map from `below(size)`, which is defined as $\{\texttt{i} : \texttt{nat} | \texttt{i} < \texttt{size}\}$, to itself. The PVS type system can thus express the specification of a function using the type system. The PVS typechecker generates proof obligations called type correctness conditions (TCCs) corresponding to subtype constraints and termination. These proof obligations can be discharged through automated proof strategies or interactively developed proofs. The PVS theorem prover combines powerful back-end automation in the form of simplification, rewriting, and satisfiability solving with user-defined proof strategies allowing proofs to be constructed at varying levels of automation, transparency, and user control.

3.1 Formal Specification of a Chart Parser

Our PVS development of the PEG chart parser takes place in a single theory titled `pegtopdown`. The first part of the theory defines a few basic types and a single constant. The type `byte` is a subtype of natural numbers from 0 to 255. The type `strings(len)` is a subtype `string` type (which defined in the PVS prelude library as a finite sequence) consisting of those character sequences that are of length `len`. The number of nonterminals `num_non_terminals` is specified as 255.[1] These bounds are specified in order to make the proofs highlight some of the complexity introduced in handling machine-representable integers. Also, we plan to use the PVS2C code generator to produce efficient imperative code implementing the parser. We also declare a variable `len` which we use to represent the length of the input string to be parsed. The two typing judgements assert the subtype relationships between `upto(len)` and `uint32`, and `non_terminal` and `uint8`. These yield proof obligations that need to be proved, but the typechecker uses these judgements without generating further proof obligations, for example, when an expression of type `non_terminal` is required to match the expected type `uint32`.

[1] We could have left the number of nonterminals unbounded and it would have made very little difference to the proof and the resulting programs would still be executable.

```
byte: TYPE = below(256)
strings(len: uint32): TYPE
  = {s : string | s`length = len}
num_non_terminals: byte = 255
non_terminal: TYPE = below(num_non_terminals)
len: VAR uint32
JUDGEMENT upto(len) SUBTYPE_OF uint32
JUDGEMENT non_terminal SUBTYPE_OF uint8
```

Fig. 5: Basic type definitions.

The entry type for the scaffold (Figure 6) is an algebraic datatype with constructors `fail` (with accessor `dep` representing the *nesting depth* (or height) and recognizer `fail?`), `pending` (with no accessors and recognizer `pending?`), `loop` (with recognizer `loop?`), `good` (with accessors `dep` and `span` and recognizer `good?`), and `push` (with accessors `pos` and `nt`, and recognizer `push?`). The values of type `ent` are entries in the scaffold. For example, the entry `fail` at position i and *nonterminal* n represents a failed parse for nonterminal n at position i.

```
ent: DATATYPE
 BEGIN
  fail(dep: uint64): fail?
  pending: pending?
  loop: loop?
  good(dep: uint64, span: uint32): good?
  push(pos: uint32, nt: uint8): push?
 END ent
```

Fig. 6: The entry type for the scaffold.

The type of PEG expressions is defined in Figure 7 by a datatype `peg`. The constructor `epsilon` represents the empty string, `failure` always parses to a failure, `any(p)` matches an input character satisfying the predicate `p`, `terminal(a)` matches the character `a`, the concatenation operation `concat`$(\mathtt{n}_1, \mathtt{n}_2)$ only allows nonterminals as the sub-grammars, and similarly for order choice `choice(A)`, the lookahead check `check(A)` (which corresponds to the & operation), and the negation operation `neg(A)`.

Figure 11 defines the type `lang_spec` of grammars, i.e., the set of productions, as a map from the type `non_terminal` to `peg`.

The predicates defined in Figure 9 capture stronger checks on the entries in the scaffold. These checks are not built into the definition of `ent` since they require the context of the length of the string and the position of the entry

```
peg : DATATYPE
 BEGIN
  epsilon : epsilon?
  failure : failure?
  any(p : [char -> bool]) : any?
  terminal(a: char) : terminal?
  concat(e1, e2: non_terminal) : concat?
  choice(e1, e2: non_terminal) : or?
  check(e: non_terminal) : and?
  neg(e: non_terminal) : not?
 END peg
```

Fig. 7: The PEG syntax

```
lang_spec: TYPE = [non_terminal -> peg]
G: VAR lang_spec
```

Fig. 8: Language Specification Type

within the scaffold.[2] The predicate `good_good_entry?` checks that `i + j` is no greater than `len` for an entry of the form `good(d, i)` with span length `i` at position `j` in the scaffold. The predicate `good_push_entry?` checks for any entry of the form `push(i, n)` in the scaffold that `i` $\leq$ `len` (since the position where `i` = `len` also has a scaffold entry) and `n` $\leq$ `num_non_terminals` (since `n` = `num_non_terminals` is used to mark the end of the stack). A stronger predicate `fine_push_entry?` checks for `push(i, n)` that `n` $<$ `num_non_terminals` indicating that it is a proper stack entry and not the end of the stack. We represent the stack such that if the top of the stack is at the position (i_0, n_0) of the scaffold A, then either n_0 = `num_non_terminals` and the stack is empty (and i_0 is irrelevant), or $A(i_0)(n_0)$ = `push`(i_1, n_1), and (i_1, n_1) is the next element in the stack. Later on, we introduce further conditions to ensure that the stack is well-formed. The predicate `nice_entry?` checks that entries of the form `good(d, i)` and `push(i, n)` satisfy `good_good_entry?` and `good_push_entry?`, respectively.

The predicate `loop_or_push?` checks that the entry is of the form `loop` or `push(i, n)`. This predicate is needed in order capture an invariant for the parsing algorithm.

With the types and predicates defined above, Figure 10 defines the type of a scaffold as an array over `upto(len)` of arrays over `non_terminal` where each entry is meets the `nice_entry?` condition.

Next, we need to characterize a well-formed stack as embedded within the scaffold. We first tried to economize by not placing any ======= The type `lang_spec` of grammars, i.e., the set of productions, is simply a map from the type `non_terminal` to `peg`.

[2] One could define `ent` as a parameterized datatype, but this has the messy consequence that the individual entries in the scaffold would be from different instantiations of the datatype.

```
good_good_entry?(len, (pos: upto(len)))
        (x: (good?)): bool
  = (pos + span(x) <= len)

good_push_entry?(len)(x: ent): bool
  = (push?(x) AND pos(x) <= len AND
   nt(x) <= num_non_terminals)

fine_push_entry?(len)(x: ent): bool
  = good_push_entry?(len)(x) AND
   nt(x) < num_non_terminals

nice_entry?(len, (pos: upto(len)))(x: ent): bool
  = (good?(x) => good_good_entry?(len, pos)(x))
   AND (push?(x) => good_push_entry?(len)(x))

m, n: VAR non_terminal

loop_or_push?(e: ent): bool
  = (loop?(e) or push?(e))

push_or_pending?(entry: ent): bool
  = push?(entry) OR pending?(entry)
```

Fig. 9: Predicates for checking the scaffold entries.

```
scaffold(len) : TYPE
  = [pos: upto(len) ->
       [non_terminal ->
        (nice_entry?(len, pos))]]
```

Fig. 10: The scaffold type

```
lang_spec: TYPE = [non_terminal -> peg]
G: VAR lang_spec
```

Fig. 11: Language Specification Type

Next, we need to characterize a well-formed stack. In an earlier attempt, we used a stack datatype that we could ensure was well-formed. We first tried to economize by not placing any constraints other than identifying stack entries $(\texttt{i}, \texttt{n})$ in the scaffold A as those where $A(i)(n)$ was of the form `push(i', n')`. Unfortunately, the interactive proofs quickly made it clear that if a stack entry points to a non-stack entry, we terminate in a state where some scaffold entries that are marked as being on the stack have not yet been processed. We could insist that every entry of the form `push(i', n')` in the scaffold A, it should be the case that $A(i')(n')$ is also of the form `push(i'', n'')`. However, this

condition can hold when there are cycles and it would be negated when any entry in the cycle is updated to a `fail`, `loop`, or `good` value. Occasionally, these experiments do succeed, and either way, we learn something from both failure and success with a modest amount of manual effort.

The predicate `successor` defined in Figure 12 checks if `entry2` is a successor of `entry1` for two stack entries `entry1` and `entry2`. Given a grammar `G` and a scaffold `A`, the `successor` predicate checks that `entry2` corresponds to a subquery of the query `entry1`. For example, if `entry1` is of the form `push(p1, nt1)` and `entry2` is of the form `push(p2, nt2)`, and `G(nt1)` is of the form `concat(n1, n2)`, then either `entry2` corresponds to the `n1` successor in which case `p1` = `p2`, or `A(p1)(n1)` has a `good` entry, and `entry2` is the `n2` successor of `entry1` so that `p2` = `p1` = `span(A(p1)(n1))` and `nt2` = `n2`. The other cases of the definition are similar.

```
successor(len, G, (A: scaffold(len)))
     (entry1, entry2: (good_push_entry?(len))): bool
 =
 (LET p1 = pos(entry1),
    nt1 = nt(entry1),
    p2 = pos(entry2),
    nt2 = nt(entry2)
   IN nt1 >= num_non_terminals
    OR CASES G(nt1) OF
      concat(n1, n2): (p2 = p1 AND nt2 = n1) OR
              (good?(A(p1)(n1)) AND nt2 = n2
              AND p2 = p1 + span(A(p1)(n1))),
      choice(n1, n2): (p2 = p1 AND nt2 = n1) OR
              (fail?(A(p1)(n1)) AND
              nt2 = n2 AND p2 = p1),
      check(n1): (nt2 = n1 AND p2 = p1),
      neg(n1): (nt2 = n1 AND p2 = p1)
      ELSE FALSE
      ENDCASES)
```

Fig. 12: The successor relation on stack entries

We use the `successor` relation to characterize a good stack within a scaffold. The recursive predicate `good_stack?` take a grammar `G`, a scaffold `A`, and an entry `stack` that is required to satisfy the `good_push_entry?` predicate. It also takes a `depth` parameter to ensure that the recursion terminates. When `stack` is empty, i.e., `nt(stack)` $\geq$ `num_non_terminals`, the `depth` must be 0. Otherwise, and in this case $0 \leq$ `nt(stack)` $<$ `num_non_terminals`, we can look up `A(pos(stack))(nt(stack))` and bind it to `entry`. We check that `entry` satisfies the `good_push_entry?` predicate, the successor relation holds between `entry` and `stack` when `entry` is nonempty, the `depth` is positive, and `entry` also recursively satisfies the `good_stack?` predicate. The well-founded termination mea-

sure is given by the `depth` parameter which decreases by one in the recursive invocation.

```
good_stack?(len, G, depth, (A : scaffold(len)))
      (stack: (good_push_entry?(len))):
  RECURSIVE bool =
  (IF nt(stack) >= num_non_terminals
   THEN depth = 0
   ELSE LET entry = A(pos(stack))(nt(stack))
      IN good_push_entry?(len)(entry)
        AND (nt(entry) = num_non_terminals OR
          successor(len, G, A)(entry, stack))
          AND depth > 0 AND
           good_stack?(len, G, depth - 1, A)
                 (entry)
  ENDIF)
  MEASURE depth
```

Fig. 13: Checking that the stack is well-formed.

For loop detection, we need to be able to check membership in the stack. This is done by the `mem_stack?` predicate whose definition looks similar to that of `good_stack?`. The `mem_stack?` predicate is only employed in the specification and is not used in defining the parsing state machine.

```
mem_stack?(len, G, depth, (A : scaffold(len)))
     (entry: (fine_push_entry?(len)),
      stack: (good_stack?(len, G, depth, A)))
  : RECURSIVE bool
 = (IF nt(stack) >= num_non_terminals
   THEN FALSE
   ELSE entry = stack OR
     mem_stack?(len, G, depth - 1, A)
          (entry, A(pos(stack))(nt(stack)))
  ENDIF)
  MEASURE depth
```

Fig. 14: Checking membership in the stack.

The `mem_stack?` predicate is used to define the predicate `fine_stack?` which checks that there are no duplicate elements in the stack. It turns out there can be no duplicate elements in a stack if it satisfies the `good_stack?` predicate since it can be shown that each element of the stack occurs at a unique depth. However, maintaining the `fine_stack?` condition as defined is simpler than proving that it is already entailed by `good_stack?`.

```
fine_stack?(len, G, depth, (A: scaffold(len)))
        (stack: (good_stack?(len, G, depth, A)))
  : RECURSIVE bool =
  (IF nt(stack) >= num_non_terminals
   THEN TRUE
   ELSE
   LET rest = A(pos(stack))(nt(stack))
    IN NOT mem_stack?(len, G, depth - 1, A)
             (stack, rest)
     AND fine_stack?(len, G, depth - 1, A)
              (A(pos(stack))(nt(stack)))
  ENDIF)
  MEASURE depth
```

Fig. 15: A stack with no repeating elements

Having dealt with the stack portion of the scaffold, we are now ready to characterize the validity conditions on the other non-`pending` entries in the scaffold. The first of these, `loop_ready?`, defined in Figure 16, checks if the cell at position `i` and nonterminal `n` is a candidate that could be marked as a loop. Essentially, a cell is `loop_ready?` if it has a successor that is either marked as a `loop` or is on the stack, i.e., is a `push` entry.

```
loop_ready?(len, G, (A: scaffold(len)),
          (i: upto(len)), n)
 : bool =
  (CASES G(n) OF
   concat(n1, n2):
    loop_or_push?(A(i)(n1)) OR
    (good?(A(i)(n1)) AND
    loop_or_push?(A(i + span(A(i)(n1)))(n2))),
   choice(n1, n2): loop_or_push?(A(i)(n1)) OR
           (fail?(A(i)(n1)) AND
           loop_or_push?(A(i)(n2))),
   check(n1): loop_or_push?(A(i)(n1)),
   neg(n1): loop_or_push?(A(i)(n1))
   ELSE FALSE
  ENDCASES)
```

Fig. 16: Condition under which a cell can be marked as a loop

The `good_fail?` predicate checks the parse should fail at position `i` of the input string for nonterminal `n`. For example, The parse could fail because `G(n)` is of the form `any(p)` and `p` fails on the character at position `i` of the input `s`. In another example, if `G(n)` has the form `concat(n1, n2)`, then the parse could fail at `i` for `n1`, or at `i + span(A(i)(n1))` for `n2`. The `fail` entry also tracks

the nesting depth of the failure. We omit the definition of `good_fail?` since it is similar to that of `good_good?` below.

The `good_good?` predicate checks that `A(i)(n)` can be marked with `good(d, sp)`. The case analysis is on the grammar given by `G(n)`. For example, if `G(n)` is `epsilon`, then the entry must be `good(0, 0)`. If `G(n)` is `choice(n1, n2)`, then either `A(i, n1)` is `good(d1, sp)`, with $\texttt{d} = 1 + \texttt{d1} + 1$, or $\texttt{A(i)(n1)} = \texttt{fail(d1)}$ and $\texttt{A(i)(n2)} = \texttt{good(d2, sp)}$, with $\texttt{d} = 1 + \texttt{max(d1, d2)}$.

The `good_good?` predicate defined in Figure 17 checks that a cell `i, n`) can be marked with `good(d, sp)`. The case analysis is on the grammar given by `G(n)`. For example, if `G(n)` is `epsilon`, then the entry must be `good(0, 0)`. If `G(n)` is `choice(n1, n2)`, then either `A(i, n1)` is `good(d1, sp1)`, and $\texttt{sp} = \texttt{sp1}$ and $\texttt{d} = 1 + \texttt{d1} + 1$, or $\texttt{A(i)(n1)} = \texttt{fail(d1)}$ and $\texttt{A(i)(n2)} = \texttt{good(d2, sp2)}$, $\texttt{sp} = \texttt{sp2}$, $\texttt{d} = 1 = \texttt{max(d1, d2)}$. The need for recording the nesting depth within the scaffold entry was discovered during the proof. Without it, we run into a problem where a scaffold could mark a cell with a self-loop with a `good` entry. This would satisfy the `good_good?` condition, but there would be no way to construct a well-formed parse tree from such a loop.

```
good_good?(len, G, (s: strings(len)))
          ((A: scaffold(len)), (i: upto(len)),
           n, (d: uint64), (sp :upto(len - i)))
  : bool =
 (CASES G(n) OF
  epsilon: sp = 0 AND d = 0,
  any(p): sp = 1 AND p(s(i)) AND d = 0,
  terminal(a): sp = 1 AND s(i) = a AND d = 0,
  concat(n1, n2):
    good?(A(i)(n1)) AND
    good?(A(i + span(A(i)(n1)))(n2)) AND
    d = 1 + max(dep(A(i)(n1)),
                dep(A(i + span(A(i)(n1)))(n2))),
  choice(n1, n2):
    (good?(A(i)(n1)) AND d = 1 + dep(A(i)(n1)))
     OR (fail?(A(i)(n1)) AND good?(A(i)(n2))
     AND d = 1 + max(dep(A(i)(n1)),
                     dep(A(i)(n2)))),
  check(n1): good?(A(i)(n1)) AND sp = 0
             AND d = 1 + dep(A(i)(n1)),
  neg(n1): fail?(A(i)(n1)) AND sp = 0
           AND d = 1 + dep(A(i)(n1)),
  failure: FALSE
 ENDCASES)
```

Fig. 17: Condition allowing a cell to be marked as `good`

The checks defined by `good_fail?`, `loop_ready?`, and `good_good?` are used to define `good_entry?` in Figure 18 as a check on any entry `u` at cell (`i`, `n`) in the scaffold.

```
good_entry?(len, G, (s: strings(len)))
          (A : scaffold(len), (i : upto(len)),
           n, (u: ent))
  : bool =
 CASES u OF
  fail(d): good_fail?(len, G, s)(A, i, n, d),
  loop: loop_ready?(len, G, A, i, n),
  good(d, sp): sp <= len - i AND
         good_good?(len, G, s)(A, i, n, d, sp)
  ELSE TRUE
  ENDCASES
```

Fig. 18: Condition under which a cell can be marked as `good`, `loop`, or `fail`

With the definition of `good_entry?`, we can start to characterize a well-formed scaffold data structure as an invariant that can be preserved during parsing leading to a post-condition that establishes the correctness of the parse. On the one hand, we can simply assert that each entry `A(i)(n)` at position `i` for nonterminal `n` satisfies the predicate

$$\texttt{good\_entry?}(\texttt{len}, \texttt{G}, \texttt{s})(\texttt{A}, \texttt{i}, \texttt{n}, \texttt{A}(\texttt{i})(\texttt{n})).$$

However, one thorny issue is that as we update the entries, we also have to establish that their nesting depth is within the `uint64` subrange. We need to maintain an invariant bounding the nesting depth entries so that they can be shown to remain within the range representable in `uint64`. The nesting depth can be bounded by the cardinality of the subset of entries that are either `good` or `fail`, since these are the only entries that represent nesting depth.

The operation `scafcount` defined in Figure 19 uses the summation operation from the PVS NASAlib `reals` library to define the cardinality of a set (predicate) over the scaffold entries.

```
scafcount(len, (A: scaffold(len)),
     (P: [ent->bool]), (j: upto(len)))
 : uint64
 = sigma(0, j, LAMBDA (i: upto(len)) :
          count(num_non_terminals)(P)(A(i)))
```

Fig. 19: Computing cardinalities over the scaffold

We can use the `scafcount` operation to define cardinalities for some specific sets. For example, `pushcount` defines the cardinality of the set of `push` entries in the scaffold. The `gfcount` operation defines the cardinality of the set of `good` or `fail` entries in the scaffold.

```
pushcount(len, (A: scaffold(len))): uint64 = scafcount(len,
 A, push?, len)

good_or_fail?(entry: ent): bool = (good?(entry) OR fail?(
 entry))

gfcount(len, (A: scaffold(len))): uint64 = scafcount(len, A,
  good_or_fail?, len)
```

Fig. 20: Computing cardinalities over the scaffold of push entries, and good or fail entries

The predicate `good_tscaffold?` defined in Figure 21 captures the correctness of the scaffold data structure (ignoring the stack) by asserting that each entry satisfies the `good_entry?` predicate, and any nested depth field in an entry is bounded by the cardinality of the set of `good` or `failed` nodes in the scaffold.

```
good_tscaffold?(len, G, (s: strings(len)))
        (A : scaffold(len)): bool =
 (FORALL (i: upto(len)), n:
  good_entry?(len, G, s)(A, i, n, A(i)(n)) AND
  (good_or_fail?(A(i)(n)) =>
    dep(A(i)(n)) <= gfcount(len, A)))
```

Fig. 21: Main Scaffold Invariant

We want to make sure that all the `push` entries in the scaffold are part of stack. For this to hold, the depth of the stack must be identical to the cardinality of the set of `push` entries in the scaffold as captured by the `good_depth?` predicate defined in Figure 22.

```
good_depth?(len, (A : scaffold(len)))
        (depth: uint64): bool
 = (pushcount(len, A) = depth)
```

Fig. 22: Depth Invariant

How do we know that when the parser has completed its work and the finalized entry for the root query has been entered? By insisting, as the predicate `good_root?` does that `A(rootpos)(rootnt)`, for the root position `rootpos` and root nonterminal `rootnt`, is never `pending`, we know that this entry in the scaffold is either on the stack or it is finalized. Parsing terminates when the stack is empty so the root entry must be a finalized entry.

```
good_root?(len, (A: scaffold(len)))
        ((rootpos: upto(len)),
        rootnt : non_terminal)
 : bool =
 (NOT pending?(A(rootpos)(rootnt)))
```

Fig. 23: The root query must be on the stack if the stack is nonempty

The final invariant on the scaffold data structure is defined in `fine_scaffold?` as a conjunction of `good_tscaffold?` and `good_root?`.

```
fine_scaffold?(len, G, (s: strings(len)))
          ((rootpos : upto(len)),
          (rootnt: non_terminal))
          (A : scaffold(len))
  : bool =
  good_root?(len, A)(rootpos, rootnt) AND
  good_tscaffold?(len, G, s)(A)
```

Fig. 24: A fine scaffold has all good entries and a good root

3.2 Defining the Parser Interpreter

We now have almost everything we need to characterize the state of our state machine. The type is specified as a dependent record with fields:

1. `scaf` for the scaffold which is typed to satisfy the predicate `fine_scaffold?`.[3]
2. `depth` for the stack depth which is typed to satisfy the predicate `good_depth?`.
3. `stack` which is the entry corresponding to the scaffold location for the top of the stack, and typed to satisfy the `fine_stack?` predicate.
4. `lflag`, a loop detection flag that is `TRUE` when a loop has been detected. The type constraint on `lflag` is that the top of the stack must satisfy `loop_ready?` when a loop has been detected, and there must be no `loop` entries in the scaffold, otherwise.

[3] If `p?` is a predicate on type `T`, then `(p?)` is short for `{x : T | p?(x)}`.

One important point about the dependent typing of the `state` record type is that the field types are carefully staged so that each field can be typed in terms of the fields that precede it. In particular, the scaffold can be updated independently.

```
state(len, G, (s: strings(len)),
   (rootpos: upto(len)),
   (rootnt: non_terminal))
  : TYPE
  = [# scaf: (fine_scaffold?(len, G, s)
               (rootpos, rootnt)),
     depth: (good_depth?(len, scaf)),
     stack: (fine_stack?(len, G, depth, scaf)),
     lflag: {b: bool |
         IF b
         THEN nt(stack) = num_non_terminals OR
            loop_ready?(len, G, scaf,
                      pos(stack), nt(stack))
         ELSE (FORALL (i: upto(len)), n:
                 NOT loop?(scaf(i)(n)))
         ENDIF}
    #]
```

Fig. 25: Parser State Type

We can define a predicate `empty?` that checks the emptiness of a stack entry, and use it to define a subtype `ne_state` of the `state` type separating those states with nonempty stacks. The parser is always invoked with a nonempty stack containing the root query.

```
empty?(stack): bool
 = (nt(stack) = num_non_terminals)

ne_state(len, G, (s: strings(len)),
        (rootpos: upto(len)),
        (rootnt: non_terminal))
 : TYPE
 = {st: state(len, G, s, rootpos, rootnt) |
    NOT empty?(st`stack)}
```

Fig. 26: Nonempty State Type

The `good_entry?` predicate is suitable for characterizing the entry-wise correctness of a scaffold, but a stronger predicate `fine_entry?` is needed to check an entry that will be used to update the scaffold. Note that while the scaffold is initalized with `pending` entries, they are never actually inserted into the scaffold.

All updates either finalize a parse for a nonterminal at a given input position to a `loop`, `fail`, or `good` entry, or push a new entry into the stack. We will be defining a function to compute such an entry for the current stack head consisting of position `i` and nonterminal `n`. This function computes an entry `u` which is either a finalized entry for `A(i)(n)` or of the form `push(i', n')` indicating that the new stack head will be at $(\texttt{i}', \texttt{n}')$, and the old stack $(\texttt{i}, \texttt{n})$ is pushed into this position, i.e., we set `A(i')(n')` to `push(i, n)`. The `fine_entry?` predicate on the newly computed entry `u` is a strengthened version of `good_entry?` which checks that `u` is either a `fail`, `loop`, or `good` entry according to `good_fail?`, `loop_ready?`, or `good_good`, respectively, or it is of the form `push(i', n')`, where `n'` is a nonterminal and the parent entry `A(i')(n')` in the stack is either a `push` or a `pending` entry. The case where `A(i')(n')` is `pending` arises because the parse at $(\texttt{i}', \texttt{n}')$ is a successor of $(\texttt{i}, \texttt{n})$ that has not yet been parsed. The case where `A(i')(n')` is a `push` entry arises when the successor node $(\texttt{i}', \texttt{n}')$ to $(\texttt{i}, \texttt{n})$ is already on the stack. The strengthened condition introduced in `fine_entry?` only became apparent through failed proof attempts. Experimenting with the proof is an effective way to explore how the complex web of relationships on the scaffold are affected by any updates.

```
fine_entry?(len, G, (s: strings(len)))
           (A : scaffold(len),
            (i : upto(len)), n, (u: ent))
  : bool =
  CASES u OF
   fail(d): good_fail?(len, G, s)(A, i, n, d),
   loop: loop_ready?(len, G, A, i, n),
   good(d, sp): sp <= len - i AND
          good_good?(len, G, s)(A, i, n, d, sp)
   ELSE fine_push_entry?(len)(u) AND
       push_or_pending?(A(pos(u))(nt(u)))
  ENDCASES
```

Fig. 27: Condition on an entry to be inserted into the scaffold

It turns out that the definition of `fine_entry?` is also not strong enough. This is because if the scaffold entry `A(i')(n')` is a `push` entry, then the parser would have detected this loop. Therefore, the only case when `u` is of the form `push(i', n')` is when `A(i')(n')` is `pending`. We also needed the explicit condition that $(\texttt{i}', \texttt{n}')$ is a successor of the top of the stack $(\texttt{i}, \texttt{n})$. This strengthening is also a consequence of failed proof attempts and is captured in the definition of the predicate `finer_entry?`. However, `fine_entry?` is independently useful in the proof so we retain the definition of `fine_entry?` instead of inlining it into that of `finer_entry?`.

The predicate `finer_entry?` can be used for a subtype definition for `finer_entry`.

```
finer_entry?(len, G, (s: strings(len)))
            (A : scaffold(len),
             (stack: (good_push_entry?(len))),
             (i : upto(len)), n, (u: ent))
  : bool =
   fine_entry?(len, G, s)(A, i, n, u) AND
   (push?(u) IMPLIES
      pending?(A(pos(u))(nt(u)))
      AND successor(len, G, A)(stack, u))

finer_entry(len, G, (s: strings(len)),
           (A: scaffold(len)),
           (stack : (good_push_entry?(len))),
           (pos : upto(len)),
           (nt: non_terminal))
  : TYPE =
  {u : ent | finer_entry?(len, G, s)
                       (A, stack, pos, nt, u)}
```

Fig. 28: Strengthened condition on an entry to be inserted into the scaffold

```
finer_entry(len, G, (s: strings(len)),
           (A: scaffold(len)),
           (stack : (good_push_entry?(len))),
           (pos : upto(len)),
           (nt: non_terminal))
  : TYPE =
  {u : ent | finer_entry?(len, G, s)
                       (A, stack, pos, nt, u)}
```

Fig. 29: Predicate subtype declaration for `finer_entry`

We again emphasize that the predicates on scaffold entries defined above are neither tedious nor obvious. They are crucial to the characterization of the correctness of the parser as well as to the construction of invariants that are preserved during parsing. They are constructed hand-in-hand with the definition of the parser through failed proof attempts and repeated refactoring. This iterative process is fairly painless. It usually takes only a few minutes to converge on the failing cases of the proof to discover where a definition is too strong or too weak or simply incorrect. The insights learned from these failing proofs help focus human attention on the tricky points in both the algorithms and the proof.

We finally arrive at the meat of the parser, the function `newentry` which computes a new entry for updating the scaffold, the one for which we spelled out the `finer_entry` type. The `newentry` function operates on the grammar `G`, the input string `s`, the original start position `start`, the root nonterminal `rootnt`, and the current state `St`. The return type for the new entry is defined by the `newentry` dependent type. The definition first extracts the top position

pos and nonterminal cur of the stack, and uses record field access (backquote) to look up the tail of the stack as St`scaf(pos)(cur). The definition has eight cases according to the grammar entry G(cur). When G(cur) is epsilon, the new entry is good(0, 0), indicating depth 0 and span 0 for the parse. If G(cur) is failure, then the new entry is fail(0) indicating a nesting depth of 0. When G(cur) is any(p), result is fail(0) if either the position pos is len or the string character s(pos) does not satisfy the predicate p. Otherwise, the result is good(0, 1) indicating a nesting depth of 0 and a span of length 1. The case when G(cur) is terminal(a) is similar. The remaining cases are handled separately and described below.

```
newentry(len, G, (s: strings(len)),
        (start: upto(len)),
        (rootnt: non_terminal))
        (St : ne_state(len, G, s, start, rootnt)):
  finer_entry(len, G, s, St`scaf, St`stack,
           pos(St`stack), nt(St`stack))
  =
  (LET pos = pos(St`stack),
     cur = nt(St`stack),
     rest = St`scaf(pos)(cur)
   IN CASES G(cur) OF
     epsilon: good(0, 0),
     failure: fail(0),
     any(p): IF pos = len OR NOT p(s(pos))
         THEN fail(0)
         ELSE good(0, 1)
         ENDIF,
     terminal(a): IF pos = len OR a /= s(pos)
           THEN fail(0)
           ELSE good(0, 1)
           ENDIF,
     concat(n1, n2): ...,
     choice(n1, n2): ...,
     check(n1): ...,
     neg(n1): ...
    ENDCASES)
```

Fig. 30: Top level of the newentry parsing function

When G(cur) is concat(n1, n2), we look up the entry u1 given by St`scaf(pos)(n1). If u1 is of the form fail(d1), then the result is fail(d1 + 1). If u1 is good(d1, sp), then we lookup u2 given by St`scaf(pos + sp)(n2). If u2 is fail(d2), then the result is fail(1 + max(d1, d2)) to capture the increment to the nesting depth from both d1 and d2. If u2 is good(d2, sp2), then the result is good(1 + max(d1, d2), sp + sp2) since the parse for cur at position pos is obtained

by concatenating the parses for n1 at `pos`, and `n2` at `pos + sp`. When `u2` is `pending`, we need to launch a parse query for `n2` at position `pos + sp`, and this is done by returning the entry `push(pos + sp, n2)`. The only remaining possibilities are that `u2` is `loop` or of the form `push(i, n)`. In either case, the result is `loop` since we have a previously detected loop at `pos + sp`, `n2`, or we have detected a new loop since `pos + sp`, `n2` is already on the stack. The remaining cases of the definition for `u2` are similar to ones described for `u2`.

```
(CASES St‘scaf(pos)(n1) OF
  fail(d1): fail(d1+1),
  good(d1, sp):
   CASES St‘scaf(pos + sp)(n2) OF
    fail(d2): fail(1 + max(d1, d2)),
    good(d2, sp2): good(1+max(d1, d2), sp + sp2),
    pending: push(pos + sp, n2)
    ELSE loop
   ENDCASES,
  pending: push(pos, n1)
  ELSE loop
 ENDCASES)
```

Fig. 31: `newentry` definition for `concat(n1, n2)`

The definition in Figure 32 for the case when `G(cur)` is `choice(n1, n2)` is quite similar.

```
(CASES St‘scaf(pos)(n1) OF
  fail(d1):
   CASES St‘scaf(pos)(n2) OF
    fail(d2): fail(max(d1,d2)+1),
    good(d2, sp): good(max(d1, d2) + 1, sp),
    pending: push(pos, n2)
    ELSE loop
   ENDCASES,
  good(d1, sp): good(d1+1, sp),
  pending: push(pos, n1)
  ELSE loop
 ENDCASES)
```

Fig. 32: `newentry` definition for `choice(n1, n2)`

We omit the definitions for the case where `G(cur)` is either `check(n1)` or `neg(n1)` since these are similar to those we have already described.

The `newentry` function forms the core of the step function `step` for our parsing state machine defined in Figure 35. The state machine halts when the

```
(CASES St`scaf(pos)(n1) OF
  fail(d1): fail(d1+1),
  good(d1, sp): good(d1+1, 0),
  pending: push(pos, n1)
  ELSE loop
 ENDCASES)
```

Fig. 33: `newentry` definition for `check(n1)`

```
(CASES St`scaf(pos)(n1) OF
  fail(d1): good(d1+1, 0),
  good(d1, sp): fail(d1+1),
  pending: push(pos, n1)
  ELSE loop
 ENDCASES)
```

Fig. 34: `newentry` definition for `neg(n1)`

stack component of the state is empty. Otherwise, we compute the `newentry` using the state `St`. If `newentry` is of the form `push(p, n)`, then we push this entry into the stack. Otherwise, `newentry` is a finalized parse entry that we use to update the scaffold, and pop the stack. The loop detection flag `lflag` is set when `newentry` is `loop`. The step function `step` is the body of a tail recursive `parse` function. The termination measure for the parse operation is a double lexicographic value defined by `size` (whose definition we omit): either the number of non-finalized cells in the scaffold decreases, or it remains the same and the stack size grows up to a bound.

The state returned by the parser is always empty.

The parse function `parse` shown in Figure 37 invokes the step function until the stack is empty, resulting in an empty state of `endstate` type (whose definition is omitted). The gateway function to the parser `doparse` is defined in Figure 38 to initialize the state and invoke the `parser` operation.

3.3 Proving Correctness

We now discuss the correctness proof. The correctness criteria are essentially embedded in the type signatures of the operations. For example, the definition of the `endstate` type ensures that all the entries in the scaffold are well-formed, i.e., satisfy the chart constraints for the given grammar. Since the stack is empty, all the loops are actual loops, and the root query has a finalized entry. There are two ways of capturing correctness through the type signature. One approach uses the declared type signature of the operation as has been done with `newentry`, `parse`, and `doparse`. The other approach employs the typing judgement mechanism in PVS to introduce a new type signature for already declared operations and expressions. We have only made sparing use of typing judgements here since

```
step(len, G, (s: strings(len)),
    (start: upto(len)), (rootnt: non_terminal))
    (St : state(len, G, s, start, rootnt))
  : state(len, G, s, start, rootnt)
 = (LET scaf = St`scaf,
        stack = St`stack,
        depth = St`depth,
        lflag = St`lflag
   IN
  IF empty?(stack)
  THEN St
  ELSE
   LET pos = pos(stack),
       cur = nt(stack),
       rest = scaf(pos)(cur),
       newentry = newentry(len, G, s,
             start, rootnt)(St)
    IN CASES newentry OF
      push(p, n): St WITH [`scaf(p)(n) := stack,
                           `stack := newentry,
                           `depth := depth + 1]
      ELSE St WITH [`scaf(pos)(cur) := newentry,
                    `stack := rest,
                    `depth := depth - 1,
                    `lflag := loop?(newentry)]
     ENDCASES
  ENDIF)
```

Fig. 35: The parsing state machine step function

```
 endstate(len, G, (s: strings(len)), (rootpos: upto(len)), (
 rootnt: non_terminal)): TYPE
 = {st: state(len, G, s, rootpos, rootnt) |
   empty?(st`stack)}
```

Fig. 36: The termination state

both the basic well-formedness of the parsing algorithm and termination rely heavily on the invariants used in the correctness arguments.

The correctness proof is constructed by discharging the subtyping and termination type correctness conditions (TCCs) which are the proof obligations generated by the PVS typechecker. The newentry operation along generates fifty nine TCCs. A few of these are mostly trivial bounds constraints on those parameters that are declared as `uint8`, `uint32`, or `uint64`, but even in some of these cases, the proofs are nontrivial. Twenty eight of the fifty nine TCCS for `newentry` are those verifying that the entry `u` constructed by each of the twenty eight cases of the definition of `newentry` satisfies the `finer_entry?` predicate.

```
parse(len, G, (s: strings(len)),
      (start: upto(len)), (root: non_terminal))
     (St : state(len, G, s, start, root))
  : RECURSIVE endstate(len, G, s, start, root)
 = (IF St`depth = 0
   THEN St
   ELSE parse(len, G, s, start, root)
             (step(len, G, s, start, root)(St))
   ENDIF)
 MEASURE size(len, G, s, start, root)(St) BY <
```

Fig. 37: The parsing engine

```
doparse(len, G, (s: strings(len)), n): ent
  = (LET pend = (LAMBDA n: pending),
     St: state(len, G, s, 0, n) = (#
     stack := push(0, n),
     depth := 1,
     scaf := (LAMBDA (i: upto(len)): pend)
         WITH
        [(0)(n) := push(0, num_non_terminals)],
     lflag := FALSE #)
   IN parse(len, G, s, 0, n)(St)`scaf(0)(n))
```

Fig. 38: The gateway parsing operation

The proofs, which are highly reusable, combine a few key lemmas and heavy use of the built-in satisfiability modulo theories (SMT) decision procedures. One such key lemma shown in Figure 39 captures how the `good_tscaffold` condition is preserved through an update with a new entry.

```
good_good_tscaffold: LEMMA
 (FORALL (s: strings(len)), (A: scaffold(len)),
        (pos: upto(len)), (nt: non_terminal),
        (u: (nice_entry?(len, pos))):
  push_or_pending?(A(pos)(nt)) AND
  good_tscaffold?(len, G, s)(A) AND
  fine_entry?(len, G, s)(A, pos, nt, u) AND
  (FORALL (j: upto(len)), m: loop?(A(j)(m))
   IMPLIES
    loop_ready?(len, G, A, pos, nt))
  IMPLIES
  good_tscaffold?(len, G, s)
                (A WITH [(pos)(nt):= u]))
```

Fig. 39: Key lemma for invariant preservation

PVS proofs are robust since they rely on a high degree of guided automation. They can be easily edit and reused on similar proof obligations, or transformed into proof strategies that are effective on a broad class of proof goals. The flexibility in the level of automation and transparency allows users to interact when needed in order to diagnose and fix problematic definitions, theorem statements, and proofs. User attention is also needed to identify patterns that are common to the proofs so that they can be encapsulated in refactored definitions, new lemmas, and proof strategies. Iteratively developing the definitions, types, and lemmas from scratch would take on the order of three to four weeks. For example, developing the lemmas and proofs for discharging the TCCs for the flat version of the parser took around week, whereas the refactored version using `newentry` required only a couple of days since it reused the proofs from the first attempt. The proof required seventeen supporting lemmas and only a handful of these needed nontrivial interaction. A finite-model counterexample capability similar to that in Alloy [**?**], Ivy [**?**], or Nitpick [**?**] might be useful but not essential since it is typically quite easy to diagnose the issue from a failed proof branch.

4 Conclusions

The present exercise is a step in the direction of building proof infrastructure for the robust and efficient verification of practical parsing algorithms. We have successfully tested the parser on several examples (including those shown in Section 2) using the PVS2CL Common Lisp code generator and the PVSio read-eval-print loop. We plan to continue polishing the proof and extending it to PEGs with data-dependent grammars.

We started our formalization attempt on a bottom-up, right-to-left scaffolding automaton parser where the scaffold entries are filled in by scanning the scaffold from the last input position to the start position, and scanning the nonterminals at each input position. A nonterminal at a given position might trigger a sub-query on another (or the same) nonterminal at the same position whose entry has not yet been finalized. We could perform a static analysis on the grammar in order to pick an ordering that ensured that there are no parsing loops and sub-queries to each query are finalized ahead of the query. Another option is to use the nonterminals in their numeric order while detecting cycles dynamically. We chose the second approach since there is no effective check for whether a PEG is well-formed, and loop detection poses an interesting verification challenge. For dynamic loop detection, one could repeatedly scan the nonterminals at each position in the scaffold until a fixpoint has been reached so that any entries that have not been finalized can be classified as loops. Alternately, one could use a hybrid approach of combining a scan with a stack of queries for those sub-queries that have not yet been finalized. The stack can also be used for loop detection.

If we are going to employ a query stack, then a top-down parser makes more sense since it does not compute irrelevant scaffold entries. We defined a top-down parser with state that contained both the scaffold and an explicit stack, and

quickly observed that the stack could be built into the scaffold data structure. This yields a significant efficiency in loop detection: an entry lookup versus a scan through an external stack. That led to the top-down chart parser described in this paper. After some preliminary experiments, we arrived at the definition of the `state` type and the dependency ordering of the fields within the state type. We first defined a flat `step` function which we could then refactor by introducing the `newentry` definition and discharging the resulting proof obligations. There were some subtleties with the type signature for the `newentry` function since it constructs two kinds of entries: the finalized ones and the new queries/stack entries. We did succeed in finding an elegant uniform characterization of the type signature using the `finer_entry?` predicate. By supporting efficient proof exploration and engineering, a proof assistant leverages the power of typechecking, proof construction and certification, and code generation to serve as an effective problem-solving aid.

The proofs we have completed are by no means the last word. There is definitely room to squeeze out more robustness and automation through the introduction of auxilliary functions, lemmas, typing judgements, and customized proof strategies. Eventually, we hope that verifying a complex parsing algorithm will be a largely automatable task as we understand how the parsing metatheory can be encapsulated into types and proof strategies. We also plan to extend the basic PEG grammar with inherited and synthesized attributes, and constraint checking, to capture some of the features the Parsley data format description language.

The present exercise is a step in the direction of building proof infrastructure for the robust and efficient verification of practical parsing algorithms. The definitions have been written in a form that is friendly to generating efficient C code. Since we are still polishing the proof, we have not currently had the time to thoroughly test the generated C code. We have successfully tested the parser using the PVS2CL Common Lisp code generator and the PVSio read-eval-print loop.