Reachability and Error Diagnosis in LR(1) Parsers

François Pottier
Inria, Paris, France
Francois.Pottier@inria.fr

Abstract

Given an LR(1) automaton, what are the states in which an error can be detected? For each such "error state", what is a minimal input sentence that causes an error in this state? We propose an algorithm that answers these questions. This allows building a collection of pairs of an erroneous input sentence and a (handwritten) diagnostic message, ensuring that this collection covers every error state, and maintaining this property as the grammar evolves. We report on an application of this technique to the CompCert ISO C99 parser, and discuss its strengths and limitations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Parsing

Keywords Compilers, parsing, error diagnosis, reachability

1. Introduction

LR parsers are powerful and efficient, but traditionally have done a poor job of explaining syntax errors. Although it is easy to report where an error was detected, it seems difficult to explain what has been understood so far and what is expected next.

This is due in part to the fact that an LR parser is fundamentally a non-deterministic machine. Several possible interpretations of what has been read so far are explored in parallel, and with each such interpretation, comes a view of what is expected next.

This is due also to the fact that information about possible pasts and futures is partly *static* (encoded in the current state of the automaton) and partly *dynamic* (encoded in its stack). When constructing an error message, static information is easy to exploit, whereas dynamic information seems more difficult to exploit, as it is not obvious how to walk the stack and summarize its contents.

Jeffery's Approach It seems tempting to select an error message based purely on static information, that is, based solely on the state in which the error is detected, regardless of the stack. Jeffery [9] describes such an approach. He suggests manually setting up and maintaining a collection of erroneous input sentences, each of which is accompanied with a diagnostic message. From this data, his tool, `merr` [8], automatically produces a mapping of states to diagnostic messages[1].

[1] Jeffery argues in favor of considering not only the current state, but also the next input symbol. However, in his Unicon parser, he seems to make little use of this feature. We restrict our attention to just the current state.

This collection of erroneous input sentences should ideally be *correct* (i.e., every sentence is erroneous), *irredundant* (i.e., no two sentences lead to the same state), and *complete* (i.e., every state where an error can occur is reached by some sentence).

Enforcing correctness and irredundancy is straightforward, and indeed, `merr` offers these features. However, it offers no support for achieving completeness, or determining whether completeness has been achieved. In fact, `merr` is independent of the parser generator, and does not have access to the grammar or automaton. All it does is run the generated parser and find out in what state the parser fails.

Faced with this problem, Jeffery advocates manually "growing" the collection over time. The collection is initially built by an expert, who studies the grammar and the automaton, and grown as end users report erroneous inputs that are not covered.

In this paper, we improve on Jeffery's approach by providing a way of enforcing completeness. Furthermore, we experimentally evaluate this approach via a real-world case study.

Improving Jeffery's Approach We believe that it is important to have automated ways of (1) initially building a complete collection of erroneous sentences and (2) testing whether a collection of erroneous sentences is complete. We leave it up to the human expert to write the accompanying diagnostic messages and (if needed) to alter the grammar so as to make errors easier to explain.

In order to provide the facilities described above, we need the ability to produce a complete set of error states, together with input sentences that cause an error in each of these states. This in turn reduces to the problem of reachability in an LR(1) automaton, which can be stated as follows:

> **Given:** A state s and a terminal symbol z.
>
> **Find:** A minimal sentence w such that, when presented with the input wz, the automaton consumes w (leaving z unconsumed) and reaches state s. Or report that no such sentence exists.

The automaton is not necessarily canonical (see §2.5 for details), otherwise the problem would be trivial (§6). We note that there is an obvious semi-algorithm for this problem: try all sentences w in turn, by increasing order of length, until one is found that satisfies the desired property. This procedure terminates if and only if state s with lookahead symbol z is reachable. It does not solve the reachability problem.

After an informal look at the problem (§2), we present a novel algorithm that solves it (§3). We have implemented this algorithm in the Menhir parser generator [14]. We present experimental data suggesting that the algorithm, although expensive, can be applied to real-world grammars with tolerable time and space requirements.

Evaluating Jeffery's Approach With the help of this algorithm, we handcraft a complete collection of diagnostic messages for the CompCert ISO C99 parser [12]. We draw several lessons from this experience.

We find that, given a sentence w that causes an error in state s, it is a nontrivial task to come up with a correct diagnostic message, let alone a "good" one. The message must not be specific of the particular input sentence w, but must reflect all sentences that lead to an error in state s. It must explain what it means to be in state s, and nothing else. It must recall the *past* (what has been recently read) and explain what are the valid *futures* (what is expected next). A key problem is that, based on the current state alone, giving an accurate list of the valid futures is not only difficult, but in fact *impossible* in some cases.

This is a weakness of Jeffery's static approach, which does not seem to have been previously pointed out. In an LR(1) automaton, there may be certain states where, without consulting the stack, one *cannot* avoid an over-approximation in the set of valid futures. Yet, such an over-approximation sounds rather undesirable. It would seem quite strange if the parser said: "Either I expect a closing parenthesis or I expect a closing bracket, but I can't tell you which". This problem is frequent in noncanonical LR(1) automata, but also arises in canonical automata, if one wishes to describe the valid futures beyond one terminal symbol.

We explain this problem (§4) and offer two ways of working around it. One way involves transforming the grammar, and relies on Menhir's existing support for parameterized rules. The other way involves adding reduction actions to the automaton, and is made possible by a new declaration, `%on_error_reduce`, which we add in Menhir. In CompCert, we exploit both ways (§5) and successfully work around the problem: our diagnostic messages never over-approximate the set of valid futures. Thus, Jeffery's approach is workable after all, but requires care.

The dual issue, whereby a diagnostic message may under-approximate the set of valid futures, also arises, due to "spurious reductions". We explain this phenomenon (§4). Under-approximation seems difficult to avoid, but (we believe) is tolerable, and in fact profitable, in many situations.

For instance, in `missing.c` (Figure 7), on line 2, after reading an incomplete expression, one encounters an invalid token, a semicolon. Instead of listing the dozens of ways in which this expression could be continued, it seems preferable to point out that a closing parenthesis is eventually required. Our diagnostic message uses the hypothetical form "If this expression is complete, then ...". This is our conventional way of pointing out that the list of futures proposed in the message is incomplete. In comparison, `clang` 3.6.0 and `gcc` 5.3.0 achieve a similar result, but (perhaps intentionally) do not point out that there are valid futures other than a closing parenthesis.

As we worked on a list of diagnostic messages for CompCert's parser, we made a series of changes to the grammar and automaton (without affecting the language), so as to limit the number of error states and so as to ensure that an accurate diagnostic message could be written for every error state. The reachability algorithm in Menhir was a key help in understanding the landscape of the error states, in guessing which changes to the grammar would be helpful, and in maintaining the partial list of messages built so far, by pointing out which error states were not yet covered and which were covered several times. In the end, we believe that this case study provides some evidence that, with appropriate tools, Jeffery's approach to error diagnosis in LR parsers is viable.

We do not attempt to offer an exhaustive comparison between CompCert's diagnostic messages and (say) `clang`'s and `gcc`'s. Such a comparison would inevitably be lengthy and subjective. We believe it is fair to say that the quality of CompCert's diagnostic messages is now on par with that of `clang` and `gcc`. (We consider only syntax errors; type errors and semantic errors are outside the scope of this paper.) This should help dispel the myth that "LR parsers (or: generated parsers) cannot produce good syntax error messages". In fact, not only do we obtain arguably good diagnostic messages, but, furthermore, we retain the comfort of generating the parser from two separate declarative specifications (namely, the grammar and the collection of diagnostic messages).

2. Approaching the LR(1) Reachability Problem

The LR(1) reachability problem is as follows: given a state s and a terminal symbol z, determine whether the automaton can reach a configuration (s, z) where the current state is s and the first unconsumed symbol is z. As in our earlier statement of the problem (§1), one may additionally ask for a minimal sentence w such that, when presented with the input wz, the automaton consumes w and reaches the configuration (s, z). One should also define exactly what is meant by "an LR(1) automaton"; this is done in §3.

2.1 Why Solve this Problem?

Once this problem is solved, it is easy to test whether a state s is an error state (that is, whether an error can be detected in state s). Indeed, it suffices to test if there exists a symbol z such that the configuration (s, z) is reachable and this configuration causes an error. Therefore, once this problem is solved, it is easy to build a complete list of error states and to test whether a given list of error states is complete.

2.2 Why we Must Care about the Lookahead Symbol

Reachability in an ordinary finite-state automaton coincides with graph reachability in the automaton's state diagram. In that setting, the reachability problem can be solved, say, by breadth-first search.

An LR(k) automaton, too, gives rise to a state diagram. Figure 2 offers an example[2]. An edge in this diagram is labeled either with a terminal symbol $z \in \Sigma$ or with a nonterminal symbol $A \in N$.

In the restricted case where the grammar is LR(0) and every nonterminal symbol generates a nonempty language, reachability still coincides with graph reachability: that is, the LR(0) automaton can reach a state s if and only if there is a path towards s in the state diagram. Indeed, suppose there is such a path. The labels found along this path compose a sentential form $\alpha \in (\Sigma \cup N)^\star$. Let $w \in \Sigma^\star$ be some sentence generated by α. (There must be one.) It is not difficult to see that, when presented with the input w, the LR(0) automaton will transition (in several steps) from its initial state to the state s, regardless of the remaining input. Therefore, s is reachable.

In general, unfortunately, we are not interested just in LR(0) grammars and LR(0) automata. Our grammars may lie outside the class LR(1): that is, they may exhibit LR(1) conflicts. Our automata are LR(1) automata: that is, their reduction actions depend on the lookahead symbol.

In such a setting, the problem can no longer be cast as a graph reachability problem. When there is an edge from s to s' labeled with a nonterminal symbol A, we cannot simply ask: "to take the edge from s to s', which sentence w should the automaton consume?". A more sensible question could be: "to take the edge from s to s', which sentence w should the automaton consume, assuming that the input symbol following w is z?".

2.3 Why we Must Care about the First Symbol

A further complication arises when we look at a sequence of two nonterminal edges, from s through s' to s''. To choose a sentence w

[2] The grammar, shown in Figure 1, is in the syntax of Menhir [14]. The braces `{}` denote empty semantic actions. This grammar is ambiguous; precedence declarations are used to "solve" conflicts. In the state diagram, the states where a reduction is permitted are doubly circled. The diagram does not show under which lookahead hypothesis each reduction is permitted, so this is really a diagram of the underlying LR(0) automaton.

```
(* The terminal symbols. *)
%token<int> INT
%token PLUS TIMES LPAREN RPAREN EOL
(* Precedence declarations. *)
%left PLUS %left TIMES %nonassoc UPLUS
(* The start symbol. *)
%start <unit> main
%%
(* The productions. *)
main: expr EOL           {}
expr:
| INT                    {}
| LPAREN expr RPAREN     {}
| expr PLUS  expr        {}
| expr TIMES expr        {}
| PLUS expr %prec UPLUS {}
```

Figure 1. A grammar.

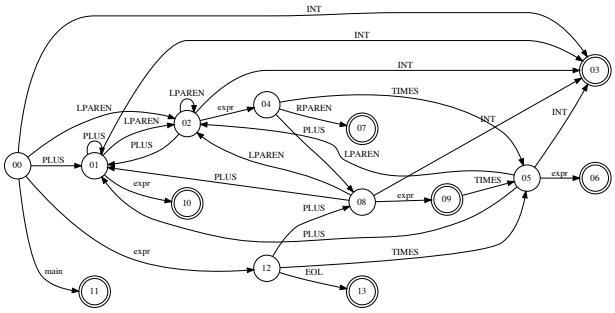

Figure 2. An LR(1) automaton for the grammar of Figure 1.

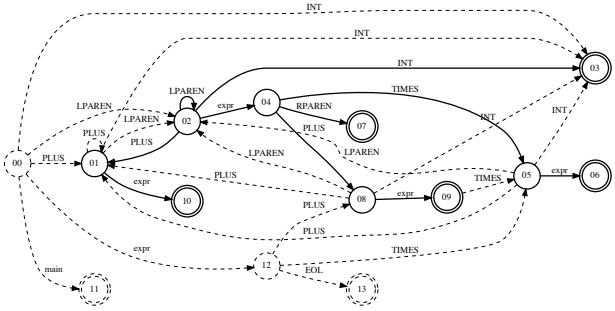

Figure 3. The star rooted at state 02 (§2.4).

that takes us through the first edge, we need to know which input symbol z follows w. Here, z is the first symbol of the sentence w' that takes us through the second edge. (More precisely, because w' could be empty, z must be the first symbol of the sentence $w'z'$, where z' is the input symbol that follows w'.) Thus, our choices of w and w' are interdependent. If somehow we have examined the first edge and chosen w and z, then the next question that we should ask is: "to take the edge from s' to s'', which sentence w' should the automaton consume, assuming that the input symbol following w' is z', and under the constraint that the first symbol of $w'z'$ must be z?" This leads us to ask a family of questions whose parameters are a nonterminal edge and *two* terminal symbols z and z'. (The *edge facts* of §3 are answers to these questions.)

2.4 Why we Must Ask about Paths

We must account for reductions. In order to take an edge labeled A from s to s', the automaton must first follow a path labeled α, where $A \to \alpha$ is a production. This path must lead from s to some state s'' where reduction is permitted (subject to a lookahead hypothesis). Reduction takes the automaton back to state s, where it follows the edge labeled A towards s'. Thus, we must ask questions not just about individual edges, but about paths: "to follow the path labeled α from s to s', which sentence w should the automaton consume, ...?" This leads us to ask a family of questions whose parameters are a *path* and two terminal symbols. (The *facts* of §3 are answers to these questions.)

For every state s, we are interested only in certain paths whose source is s. More precisely, a path labeled α, whose source is s, is of interest if (1) this path exists in the automaton and (2) s has an outgoing edge labeled A and (3) $A \to \alpha$ is a production. (Furthermore, a prefix of a path of interest is of interest too.) We refer to the set of these paths as the *star* rooted at s.

For example, Figure 3 shows the star rooted at state 02. The vertices and edges in bold are part of the star, while the dashed vertices and edges are not. By finding out how to travel through this star, we can tell how the nonterminal edge from state 02 to state 04 can be taken. Because the symbol `expr` has five productions, this star has five branches. Because a star is a set of paths, one can think of it as a tree; however, once projected onto the automaton's state diagram, it can have sharing and cycles, as in this example, where the branch that corresponds to the production `expr` → `LPAREN expr RPAREN` goes from state 02 to itself, then to states 04 and 07.

We define the size of the star rooted at s as the sum of the lengths of the paths that compose it. In Figure 3, the size of the star rooted at state 02 is 12 (the sum of the sizes of the productions for the symbol `expr`), even though, once projected onto the graph, this star involves only 10 graph edges. We define the *total star size* S as the sum of the sizes of the stars rooted at every state s. This is the number of paths of interest, as defined above. This parameter plays a role in the complexity of the algorithm (§3). An upper bound on it is $S \leq n.|\mathcal{G}|$, where n is the number of states of the automaton and $|\mathcal{G}|$ is the size of the grammar, which we define as the sum of the sizes of all productions.

2.5 Why we Must Tolerate Non-Canonical Automata

A practical parser generator does not usually build a canonical LR(1) automaton. Instead, it may (1) merge several states together, (2) introduce default reductions, (3) "solve" shift/reduce and reduce/reduce conflicts. Points (1) and (2) introduce new reduction actions, whereas (3) removes shift actions and reduction actions (possibly making certain states unreachable). Our algorithm tolerates these alterations. It accepts any LR(1) automaton that is sound, but not necessarily complete, with respect to the grammar.

3. An LR(1) Reachability Algorithm

We assume that the automaton is described by two tables, which indicate which transitions exist and which reductions are permitted. On paper, we write:

$$\mathcal{A} \vdash s \xrightarrow{c} s'$$

if there is a transition from state s to state s' labeled with the terminal or nonterminal symbol c; and we write:

$$\mathcal{A} \vdash s \text{ reduces } A \to \alpha \text{ on } z$$

if in state s it is permitted to reduce the production $A \to \alpha$ when the first unconsumed input symbol is z.

We very briefly recall the small-step dynamic semantics of LR automata. A stack σ is a nonempty sequence of states. At every

time, the automaton has a "current" stack. If the current stack is σ, then its topmost element $\mathrm{top}(\sigma)$ is considered the "current" state. When a (terminal or nonterminal) transition is taken, the target state of the transition is pushed onto the stack, and therefore becomes the current state. When a production $A \to \alpha$ is reduced, $|\alpha|$ states are popped off the stack, where $|\alpha|$ is the length of the sentential form α. Thus, the stack that existed before α was recognized is restored. Then, a transition labeled A is taken.

3.1 Specification

We begin with a specification of our algorithm, that is, a high-level description of what the algorithm is supposed to compute. The algorithm computes and accumulates *facts* of the following form:

$$s \xrightarrow{\alpha \,/\, w}\!\!\!\!\!\rightarrow\; s' \,[z]$$

Let us recall that s and s' are states; α is a sentential form, that is, a sequence of terminal and nonterminal symbols; w is a sentence, that is, a sequence of terminal symbols; z is a terminal symbol. Let us note also that, because the automaton is deterministic, the state s and the sequence of edge labels α together determine at most one path in the automaton's state diagram. We are interested in such a fact only if it is of interest, that is, only if the path labeled α out of s is of interest (§2.4). Such a fact is informally interpreted as follows:

> The path determined by s and α, which leads to s', is taken by consuming w, provided the next input symbol is z.

or, put another way:

> If the automaton is in state s and the remaining input begins with wz, then the automaton makes a series of transitions along the path α, consuming w and ending up in state s'.

This informal interpretation does not mention the automaton's stack. In order to make the meaning of a fact completely clear, let us explicitly state how the stack evolves. When the automaton makes a series of transitions along a path labeled α, the $|\alpha|$ states found along this path are pushed onto the stack. Thus, in full detail, the interpretation of a fact is:

> INTERPRETATION OF $s \xrightarrow{\alpha \,/\, w}\!\!\!\!\!\rightarrow\; s' \,[z]$.
>
> Let $s_0 \ldots s_n$ be the states found along the path labeled α out of s. (Thus, s_0 is s and n is $|\alpha|$ and s_n is s'.)
>
> For every stack σ,
> if the current stack is σ where $\mathrm{top}(\sigma) = s_0$
> and if the remaining input begins with wz,
> then w is consumed and the current stack becomes $\sigma s_1 \ldots s_n$.

It may be worth noting that s and α together determine s'. Thus, we could have chosen not to mention s' at all in our notation for facts. Yet, it seems preferable, for the sake of readability, to include it. Conversely, one may wonder whether s and s' determine α. If that were the case, then we could omit α in our notation for facts. However, this is not true: there can be several distinct paths, carrying distinct sequences of edge labels, from s to s'.

The algorithm also computes and accumulates *edge facts* of the following form:

$$s \xrightarrow{A \,/\, w} s' \,[z]$$

This looks very much like a fact, and indeed, its informal interpretation is the same: we take such an edge fact to mean that the edge labeled A from s to s' is taken by consuming the input w, provided the next input symbol is z. Yet, formally, we distinguish the "fact" and "edge fact" predicates: notice the double arrowhead in a fact, versus the single arrowhead in an edge fact. On paper, this leads to a simple (mutually inductive) characterization of these two predicates. Furthermore, in the implementation, we actually keep track of facts and edge facts in two separate data structures.

The "fact" and "edge fact" predicates can be inductively characterized: they are the least predicates that satisfy the deduction rules in Figure 4. The first three rules characterize the "fact" predicate, whereas the last rule characterizes the "edge fact" predicate.

Rule INIT asserts a zero-transition fact. It asserts that, regardless of the next input symbol z, we can go from state s to itself via an empty path and by consuming nothing. (ϵ stands for the empty word.) Rule STEP-TERMINAL extends a fact with one new transition, labeled with a terminal symbol z. It asserts that, if we can reach state s' under the assumption that the next input symbol is z, and if the automaton has a transition labeled z from s' to s'', then we can reach state s''. This holds regardless of the input symbol z' that follows z. Rule STEP-NONTERMINAL extends a fact with one new transition, labeled with a nonterminal symbol A. Its first two premises are analogous to those of STEP-TERMINAL. Some complication stems from the fact that A is a nonterminal symbol. In order to take the transition labeled A from s' to s'', we must consume an input fragment w' and beyond it see a lookahead symbol z' that cause this transition to be taken. This is expressed by the third premise, which is an edge fact. We would then like to conclude that, by consuming ww' and beyond it seeing z', we move all the way from state s to state s''. For this to be the case, one condition remains to be checked, which is expressed by the fourth premise. Indeed, the first premise contains the hypothesis that the first input symbol beyond w is z. We must ensure that this hypothesis is satisfied: hence, we must check that the first symbol of the sentence $w'z'$ is z. (We write $\mathrm{first}(w'z')$ for the first symbol of the nonempty sentence $w'z'$. This has nothing to do with the notion of a "FIRST set".) Finally, rule REDUCE spells out when an edge fact holds, that is, under which conditions a nonterminal transition can be taken. In short, we can take a transition labeled A from s to s'' if and only if (premise 2) we are able to travel from s, along a path labeled α, to some state s' where (premise 3) the production $A \to \alpha$ can be reduced.

The "fact" and "edge fact" predicates can be viewed as a big-step dynamic semantics of LR automata. In contrast with the small-step dynamic semantics, it does not mention the stack. (It need not mention it, because it treats the stack in a uniform way: the interpretation of facts begins with "For every stack σ, ...") Thus, it has a "finite-state" flavor that is of critical importance in the algorithm.

In this paper, we take it for granted that this big-step dynamic semantics is correct and complete with respect to the small-step semantics. In principle, one should prove the two semantics equivalent. We anticipate no difficulty in this proof.

The purpose of our algorithm should now begin to be apparent: roughly speaking, the algorithm computes all valid facts (and edge facts), that is, all possible ways of traveling in the automaton's state diagram. Of course, there may be (and there usually is) an infinite number of them. We limit the set of facts (and edge facts) that the algorithm gathers by introducing a *subsumption* relation on facts (and edge facts). The subsumption relation between two facts, written $f_1 \leq f_2$, means that f_1 is "better" than f_2. This implies that, if we have already discovered and recorded f_1, then we do not need to record f_2. This relation is defined as follows:

$$\textsc{Subsumption}\quad \frac{\mathrm{first}(w_1 z) = \mathrm{first}(w_2 z) \qquad |w_1| \leq |w_2|}{s \xrightarrow{\alpha \,/\, w_1}\!\!\!\!\!\rightarrow\; s' \,[z] \leq s \xrightarrow{\alpha \,/\, w_2}\!\!\!\!\!\rightarrow\; s' \,[z]}$$

(The subsumption relation between two edge facts is defined similarly.) This definition implies that two facts can be in the subsumption relation only if they concern the same path (namely, the path labeled α out of s to s'), the same lookahead hypothesis (namely, z)

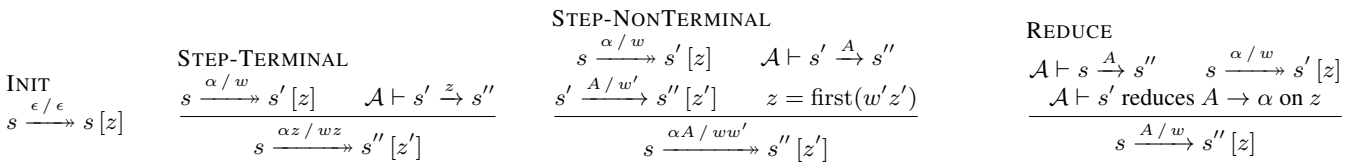

Figure 4. Inductive characterization of the "fact" and "edge fact" predicates.

and the same first input symbol (namely, $\mathrm{first}(w_1 z)$, also known as $\mathrm{first}(w_2 z)$). Under these conditions, the sentences w_1 and w_2 are just two ways of achieving exactly the same effect, so it suffices to record one of them. We record one whose length is minimal.

The purpose of the algorithm can now be reformulated in a more precise manner. The algorithm constructs a set F of facts of interest that is complete up to subsumption. In other words, if some fact f_2 is of interest and can be obtained by applying the rules of Figure 4, then F contains a fact f_1 such that $f_1 \leq f_2$ holds. Similarly, the algorithm constructs a set of edge facts E that is complete up to subsumption.

We can bound the size of the sets F and E, as follows. Thanks to subsumption, for every path labeled α out of s, for every first input symbol $\mathrm{first}(wz)$, and for every lookahead hypothesis z, the set F contains at most one fact. Furthermore, the number of paths of interest is at most S, the total star size (§2.4). Thus, we have $|F| \leq S.|\Sigma|^2$. Similarly, we have $|E| \leq m.|\Sigma|^2$, where m is the number of edges labeled with a nonterminal symbol.

3.2 Algorithm

Pseudocode for the algorithm is given in Figure 5. It follows the rules of Figure 4 quite closely, so we do not explain it in detail. In addition to the sets F and E, which have been mentioned already, the algorithm uses a priority queue Q to store a set of facts that await further examination. The priority of a fact is the length of its component w, so facts that concern shorter input sentences are examined (and, if not already known, recorded) first.

At a high level of abstraction, the algorithm can be viewed as a variant of Dijkstra's shortest paths algorithm. In Dijkstra's algorithm, there is a single source vertex, and the graph edges are fixed and known ahead of time. Here, every vertex s is a source: one could say that we are running, in parallel, one instance of Dijkstra's algorithm out of every source s. Furthermore, these instances communicate with one another: when a path to a certain vertex is discovered in one instance, a new edge may be created (indeed, this is one reading of rule REDUCE), which becomes immediately visible to all instances. Fortunately, in rule REDUCE, the weight of the newly-created edge is no less than the weight of the path that caused its creation. (Both weights are $|w|$.) This allows us to properly synchronize all instances via a single priority queue and maintain the key property that a fact, once recorded, can never be subsumed by a newly discovered fact.

The time complexity of the algorithm is clearly polynomial in $|\Sigma|$, the size of the input alphabet, n, the number of states of the automaton, and $|\mathcal{G}|$, the size of the grammar. An informal analysis suggests that it is $O(S.|\mathcal{G}|.|\Sigma|^3)$, where the total star size S is bounded by $n.|\mathcal{G}|$ (§2.4).

3.3 Implementation Details

We assume $|\Sigma| \leq 2^8$ and represent a terminal symbol as an 8-bit character and a sentence w as a character string. (This is not essential. We could remove this assumption and use lists of integers instead of strings.) On top of this, we impose maximal sharing (i.e., hash-consing) of sentences. In practice, the number of unique sentences is small. (For instance, in one run of the algorithm, 3.7 million facts were recorded in F, but the number of unique sentences w was less than 3.10^4.) We assign a unique index (a small integer) to each sentence: this allows us to encode a reference to a sentence in less than one word of memory.

We precompute the star rooted at every state s. This yields a trie, that is, a tree structure where each edge towards a child is labeled with a terminal or nonterminal symbol. The total number of trie nodes thus constructed is S, the total star size. In the grammars that we have seen, this number is under 10^5. We assign a unique index to each trie node: this allows us to encode a reference to a path of interest (that is, a triple of s, α and s') in less than one word of memory.

A fact is in principle a record of three fields: a path (s, α, s'), a sentence (w), and a lookahead symbol (z). Thanks to the encodings described above, we are able to pack this information in one 64-bit word of memory, thus avoiding the need to allocate facts as records. Compared to heap allocation of facts, this low-level encoding saves roughly a factor of two in space.

Since priorities are low nonnegative integers, a simple-minded, array-based priority queue can be used. Priorities serve as array indices. The total worst-case complexity of k insertion and k extraction operations is $O(k) + O(p)$, where p is the largest priority that is ever used. In practice, p is very small (say, 15) while k is very large (in the millions), so the amortized complexity of the priority queue operations is effectively $O(1)$. This data structure beats a binary heap (stored in an array) by a significant factor: the global impact of this choice is a factor of roughly two in execution time.

The set of facts F must support the following operations.

1. Test whether a fact $s \xrightarrow{\alpha \,/\, w}\!\!\!\!\rightarrow\!\!\!\!\rightarrow s'\,[z]$ is new (i.e., not subsumed by some fact already in F) and if so, add it to F. This is used in the main loop.

2. Given s' and z, enumerate all facts in F of the form $s \xrightarrow{\alpha \,/\, w}\!\!\!\!\rightarrow\!\!\!\!\rightarrow s'\,[z]$. This is used in procedure STEP-NONTERMINAL-RIGHT.

The set of edge facts E must support the following operations:

1. Test whether an edge fact $s \xrightarrow{A \,/\, w} s'\,[z]$ is new (i.e., not subsumed by some edge fact already in E) and if so, add it to E. This is used in procedure REDUCE.

2. Given s, A, z, z', enumerate all sentences w such that $s \xrightarrow{A \,/\, w} s'\,[z']$ is in E and $z = \mathrm{first}(wz')$. This is used in procedure STEP-NONTERMINAL-LEFT.

An efficient implementation of F and E requires a little thought, but can be built (in several ways) with off-the-shelf data structures (arrays, association maps, hash tables, bit sets).

3.4 Optimizations

We implement two optimizations that are not reflected in the pseudocode of Figure 5.

We construct and examine a fact $s \xrightarrow{\alpha \,/\, w}\!\!\!\!\rightarrow\!\!\!\!\rightarrow s'\,[z]$ only if z does not cause an error in state s', that is, only if the state s' has a (shift or reduce) action on z. It would be useless to consider a fact

procedure STEP-TERMINAL($s \xrightarrow{\alpha \,/\, w}\!\!\!\!\!\rightarrow\; s'\,[z]$)
  for each s'' **such that** $\mathcal{A} \vdash s' \xrightarrow{z} s''$ **do**
  — there is at most one such s''
    for each z' **do**
      insert the fact $s \xrightarrow{\alpha z \,/\, wz}\!\!\!\!\!\rightarrow\; s''\,[z']$ **into** Q

procedure STEP-NONTERMINAL-LEFT($s \xrightarrow{\alpha \,/\, w}\!\!\!\!\!\rightarrow\; s'\,[z]$)
  for each A and s'' **such that** $\mathcal{A} \vdash s' \xrightarrow{A} s''$ **do**
    for each edge fact $s' \xrightarrow{A \,/\, w'}\!\!\!\!\!\rightarrow\; s''\,[z']$ in E
    such that $z = \text{first}(w'z')$ **do**
      insert the fact $s \xrightarrow{\alpha A \,/\, ww'}\!\!\!\!\!\rightarrow\; s''\,[z']$ **into** Q

procedure STEP-NONTERMINAL-RIGHT($s' \xrightarrow{A \,/\, w'} s''\,[z']$)
  for each fact of the form $s \xrightarrow{\alpha \,/\, w}\!\!\!\!\!\rightarrow\; s'\,[z]$ in F
  such that $z = \text{first}(w'z')$ **do**
    insert the fact $s \xrightarrow{\alpha A \,/\, ww'}\!\!\!\!\!\rightarrow\; s''\,[z']$ **into** Q

procedure REDUCE($s \xrightarrow{\alpha \,/\, w}\!\!\!\!\!\rightarrow\; s'\,[z]$)
  if $\mathcal{A} \vdash s'$ reduces $A \to \alpha$ on z **then**
    for each s'' **such that** $\mathcal{A} \vdash s \xrightarrow{A} s''$ **do**
    — there is exactly one such s''
      let e be the edge fact $s \xrightarrow{A \,/\, w} s''\,[z]$
      if e is not subsumed by any edge fact in E **then**
        insert e **into** E
        call STEP-NONTERMINAL-RIGHT(e)

procedure REACHABILITY()
  $Q, F, E \leftarrow \emptyset$
  for each s and z **do**
    insert the fact $s \xrightarrow{\epsilon \,/\, \epsilon}\!\!\!\!\!\rightarrow\; s\,[z]$ **into** Q
  while Q is nonempty **do**
    take out of Q a fact f of minimal sentence length
    if f is not subsumed by any fact in F **then**
      insert f **into** F
      call STEP-TERMINAL(f)
      call STEP-NONTERMINAL-LEFT(f)
      call REDUCE(f)

Figure 5. The reachability algorithm.

that violates this property, as such a fact could not possibly lead to a successful reduction. In practice, this optimization reduces the number of facts that go through the priority queue by a factor of roughly two.

We note that rule STEP-TERMINAL has a terminal symbol z' in its conclusion, which does not appear in the premises. Provided its premises are satisfied, the rule can be applied to every terminal symbol z', thus producing $|\Sigma|$ similar-looking conclusions. In the implementation, instead of actually generating and recording $|\Sigma|$ distinct facts, we record just one summary fact, whose lookahead hypothesis takes the form [_]. This complicates the code, but saves

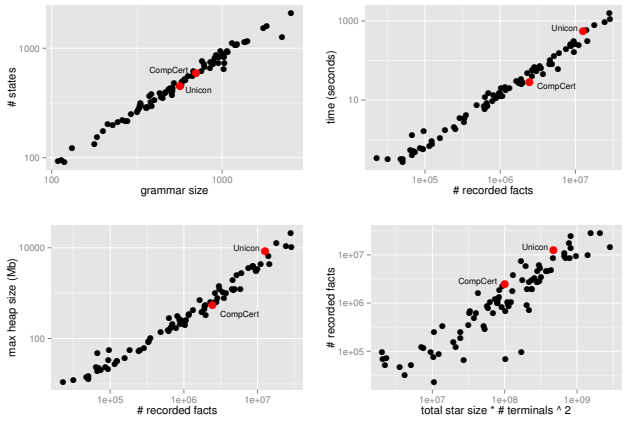

Figure 6. Performance aspects of the reachability algorithm.

time and space: we observe a reduction of a factor of up to two in the number of facts that we record.

3.5 How to Build a Complete Collection of Erroneous Inputs

Once the reachability algorithm has run, the edge facts in the set E tell us exactly how each nonterminal transition can be taken. Based on this information, we define an (implicit) graph whose vertices are pairs of a state s and a lookahead symbol z, and whose edges are labeled with sentences w. This graph has an edge labeled w from (s, z) to (s', z') if and only if either:

1. $\mathcal{A} \vdash s \xrightarrow{z} s'$ holds and w is the singleton sentence z; or
2. E contains an edge fact $s \xrightarrow{A \,/\, w} s'\,[z']$ and $z = \text{first}(wz')$.

We then run Dijkstra's shortest paths algorithm on this graph. The source vertices are (s, z), where s is an initial state. The target vertices are (s, z), where s has an error on z. If we find that a target vertex (s, z) can be reached via a (shortest) path whose labels form the sentence w, then the sentence wz is a (minimal) sentence that causes an error in state s with lookahead symbol z. If a target vertex (s, z) cannot be reached, then it is impossible to cause an error in state s with lookahead symbol z. This computation seems relatively cheap. In our measurements, it is typically 10 times faster than the reachability algorithm.

3.6 Performance Aspects

We ran the reachability algorithm (including the postprocessing phase of §3.5) on a 40-core Intel Xeon running at 2.4GHz, equipped with 1Tb of RAM[3]. We tried over 200 grammars found in Menhir's test suite, using LALR as the construction method. Of these, 85 required at least 0.25 seconds of processing time. All of these are "real-world" grammars. We measured several aspects of the algorithm's performance, a few of which appear in Figure 6. Every plot uses logarithmic scales.

The top-left plot shows that n, the number of states in the LALR automaton, is correlated with $|\mathcal{G}|$, the size of the grammar, which we define as the sum of the sizes of the productions. The estimated slope in the log-log diagram is 0.99, which shows that the correlation is linear. This experimental data confirms Purdom's findings [15].

[3] The algorithm is sequential, so does not benefit from multiple cores. By today's standards, this is a relatively slow machine. The large amount of RAM and the ability to process many grammars in parallel are our motivations for using this machine.

The top-right plot shows that the time consumption is correlated with the number of facts that are recorded in the set F upon completion. The estimated slope is 1.25: the running time grows superlinearly with the number of facts. For the most difficult grammar in our test suite, a PHP grammar, over 28 million facts are recorded, and the time requirement is 1600 seconds.

The bottom-left plot shows that the space consumption is correlated with the number of facts. The estimated slope in the log-log diagram is 1.05, so the correlation seems linear. The estimated slope in the original diagram is 5.10^{-4}, that is, about 500Mb per million facts. For instance, for Unicon, (a newer version of) the grammar considered by Jeffery [9], 12.6 million facts are recorded, and the space requirement is 8Gb. For the PHP grammar mentioned above, the space requirement is 21Gb.

The data points for CompCert and Unicon show that the size of the grammar alone cannot be used to predict the resources required by the algorithm. Unicon's grammar is smaller than CompCert's; yet, it requires 20x as much time and leads to gathering 5x as many facts. Furthermore, the time required to process CompCert's grammar varies widely depending on which version of the grammar one analyzes. The data shown here concerns the current version. We note that we suffered a 3x slowdown after we added `%on_error_reduce` declarations (§4).

The bottom-right plot shows $|F|$, the final number of facts, as a function of $S.|\Sigma|^2$. The estimated slope in the log-log diagram is 0.97, which suggests a linear correlation. This seems to echo the theoretical bound $|F| \leq S.|\Sigma|^2$. That said, the fit does not seem very good. We would like to better understand how to predict $|F|$ and (therefore) the algorithm's space and time requirements.

4. Crafting Accurate Diagnostic Messages

A diagnostic message should describe the *past* (that is, how the input so far has been interpreted) and the *future* (that is, what is expected next). Ideally, one might think, a diagnostic message should be *correct* (i.e., propose only valid futures) and *complete* (i.e., propose all valid futures). One may decide to intentionally abandon completeness, because listing all valid futures would lead to verbose and complicated messages; but (we claim) one should not inadvertently lose completeness.

Ideally, a future should be described using both terminal and nonterminal symbols: saying that "an expression is expected" is preferable to listing the dozens of terminal symbols that could be the beginning of an expression. A future should sometimes have length greater than one: saying "a comma, followed with an expression, is expected" may be preferable to saying just "a comma is expected".

In our setting, where a fixed mapping of states to diagnostic messages is established, and where the automaton is noncanonical, ensuring that diagnostic messages are correct and complete can be difficult. Indeed, the presence of spurious reductions compromises both correctness and completeness. It is worth understanding this phenomenon, not only in order to try and preserve one or both of these properties, but also because (we claim) spurious reductions can be exploited to our advantage and help produce concise and correct diagnostic messages.

Loss of Correctness Correctness is compromised as follows. Say a sentence wz leads to an error in state s, and this state has a reduction action on the terminal symbol z'. When writing a diagnostic message for the state s, one might naively decide to list z' as a possible future. Yet, it may well be the case that the sentence wz' causes an error, too! Then, this diagnostic message is incorrect. (The reduction, in this case, is spurious; the error is detected only after this reduction.)

For a concrete example of this phenomenon, take the grammar of Figure 1 and extend it with a second kind of delimiters, say, brackets. That is, add the production `expr` → `LBRACK expr RBRACK`. The LALR automaton for this extended grammar contains a state where reducing the production `expr` → `expr PLUS expr` is permitted on both `RPAREN` and `RBRACK`. Yet, the diagnostic message: "Either a closing parenthesis or a closing bracket is expected", which the user interprets as: "You may choose between a closing parenthesis and a closing bracket", is incorrect. Depending on what has been read previously, either a parenthesis is definitely expected, or a closing bracket is definitely expected, but it is never the case that both are accepted.

In a canonical automaton, if one describes a future by just one terminal symbol, then this issue does not arise, as the lookahead sets are never over-approximated. Yet, if one wishes to describe a future *beyond one terminal symbol*, then the issue arises again. For instance, in C99, imagine the parser detects an error while reading a list of *declaration-specifiers*, such as `static const`. This list could be the beginning of (1) a *declaration*, (2) a *function-definition*, or (3) a *parameter-declaration*. In order to explain what can come after this list, if it is finished, the parser needs to know (and the user expects the parser to know) in which subset of these three cases we are. For instance, if we are within a block, then `static const` must be the beginning of a declaration: we are in case $\{1\}$. On the other hand, if we are the top level, `static const` could be the beginning of a declaration or function definition: we are in case $\{1, 2\}$. Unfortunately, the answer to the question: "Are we in a block, or at the top level?" is not static, that is, not encoded in the current state of the automaton. Indeed, even a canonical automaton distinguishes only as many states as is necessary to statically tell which terminal symbols are permitted next. Here, the set of permitted terminal symbols in cases $\{1\}$ and $\{1, 2\}$ is the same, namely "variable identifier, type identifier, star, or opening parenthesis". In summary, even in a canonical automaton, knowledge of the current state is not sufficient to accurately describe what must come next, beyond the first input symbol.

Loss of Completeness Completeness is compromised as follows. Say a sentence wz' leads to an error in state s', and this state has no action on some terminal symbol z. When writing a diagnostic message for the state s, one might naively decide *not* to list z as a possible future. Yet, it may well be the case that the sentence wz does *not* cause an error! Then, this diagnostic message is incomplete. (The state s', in this case, must have been reached via wz' because of a spurious reduction; it may be the case that the sentence wz does not cause this spurious reduction and does not lead to the state s'.)

For a concrete example of this phenomenon, again take the grammar of Figure 1, extend it with brackets, as above, and further extend it so as to allow a comma-separated *list* of expressions, abbreviated as `snl(COMMA, expr)`[4], to appear between a pair of matching parentheses or brackets. We now have the productions:

`expr` → `LPAREN snl(COMMA, expr) RPAREN`
`expr` → `LBRACK snl(COMMA, expr) RBRACK`

The LALR automaton for this grammar has a state (namely, state 7) whose LR(1) items are as follows:

`expr` → `expr` • `PLUS expr`		[...]
`expr` → `expr` • `TIMES expr`		[...]
`snl(COMMA, expr)` → `expr` •		[`RPAREN`, `RBRACK`]
`snl(COMMA, expr)` → `expr` • `COMMA snl(COMMA, expr)`		[...]

In this state, reducing the production `snl(COMMA, expr)` → `expr` is permitted on both `RPAREN` and `RBRACK`. Because of this over-approximation, some errors that "should" be detected in state 7

[4] Menhir offers a parameterized nonterminal `separated_nonempty_list(sep, elem)`, which we abbreviate as `snl(sep, elem)`.

are detected elsewhere. For instance, the incorrect sentence `LBRACK INT RPAREN`, where the wrong closing delimiter is used, first leads the automaton into state 7, where a spurious reduction takes place, leading the automaton to state 5, whose LR(1) items are as follows:

`expr` → `LBRACK snl(COMMA, expr)` • `RBRACK` [...]

In state 5, the automaton "thinks" that the list of expressions is finished and that the only valid future is a closing bracket, `RBRACK`. Yet, the diagnostic message: "A closing bracket is expected" would be incomplete. Indeed, it is clear that, after reading `LBRACK INT`, the symbol `RBRACK` is not the only valid future: a complete list of permitted terminal symbols at this point is `PLUS`, `TIMES`, `COMMA`, and `RBRACK`. The symbols `PLUS` and `TIMES` are permitted because, although `INT` forms an expression, perhaps this expression is not finished. The symbol `COMMA` is permitted because, although `INT` forms a list of expressions, perhaps this list of expressions is not finished.

In summary, a spurious reduction can take the automaton into a state where the set of possible futures is under-approximated. In other words, a spurious reduction causes the automaton to commit to a certain interpretation of the past and reduces the set of permitted futures. The human expert must be aware of this phenomenon, if she wishes to avoid (or, at least, control) incompleteness.

We believe that a good diagnostic message, in this case, could be: "Up to this point, a list of expressions has been recognized. If this list is complete, then a closing bracket is expected". Such a diagnostic message explicitly proposes only one future, namely a closing bracket, but acknowledges the existence of others.

Spurious Reductions Considered Beneficial Spurious reductions are not all that bad, after all. In the previous example, a spurious reduction takes us out of state 7, where it is impossible to produce a correct list of futures (because both `RPAREN` and `RBRACK` seem legal), and (after looking up the stack) takes us to state 5, where it is possible to produce such a list (because it is evident there that `RBRACK` is legal, whereas `RPAREN` is not). By allowing a reduction to take place, we are able to exploit dynamic information (that is, we let the diagnostic message to depend on the contents of the stack), and we recover correctness. The price to pay is that in state 5, as explained above, we cannot produce a complete list of futures.

In this light, spurious reductions seem helpful. So much so, in fact, that we suggest artificially causing more of them. In the previous example, in state 7, if the invalid token is `RPAREN` or `RBRACK`, then a spurious reduction of the production `snl(COMMA, expr)` → `expr` takes place, but if the invalid token is `INT`, `LPAREN`, or `LBRACK`, then it does not: there is no action on these symbols. If reduction was allowed in these cases, too, then an error would never be detected in state 7. Instead, the automaton would go to state 5, where the error would be detected. We extend Menhir with a new declaration, `%on_error_reduce snl(COMMA, expr)`, whose effect is precisely to add the missing reduction actions to the automaton. This does not affect the language that is accepted by the automaton, but affects the set of states in which errors can be detected. For a more detailed explanation of this mechanism, see Menhir's documentation [14].

Selective Duplication Another way of recovering correctness, without resorting to spurious reductions, is to split some states, so as to make more information static. This is done by duplicating the definition of certain nonterminal symbols. We achieve this by macro-expansion, without any actual duplication in the grammar. The trick is to parameterize certain nonterminal symbols with a phantom parameter, which encodes contextual information. For instance, in C99, we equip *declaration-specifiers* with a parameter that indicates whether we are within a block, at the top level, or within a parameter declaration. As Menhir expands away parameterized nonterminal symbols, the effect is the same as if we had defined three identical copies of *declaration-specifiers*. We obtain an automaton where more states are distinguished, so that, in every error state where a list of *declaration-specifiers* has just been read, it is now possible to correctly list the valid futures. For a more detailed explanation of selective duplication, see Menhir's documentation [14].

5. Application to CompCert C

We extend Menhir [14] with several commands for: (1) producing from scratch a complete collection of erroneous sentences and diagnostic messages; (2) checking that such a collection is correct, irredundant, and complete; (3) maintaining this collection as the grammar evolves; (4) compiling this collection down to an OCaml function that maps a state number to a diagnostic message. More details can be found in the reference manual [14].

The CompCert compiler [12] contains two C parsers, both of which are constructed by Menhir. The "pre-parser", which runs first, is somewhat more complex, as it is in charge of distinguishing type names and variable names; for this purpose, it contains a "lexer hack". It is currently unverified. Because it runs first, it is in charge of detecting syntax errors. The "parser", which runs next, is based on the grammar found in the C99 standard. It is verified [10].

Using Menhir's new features, we develop diagnostic messages for CompCert's pre-parser. We find at first that, due to a lack of static information, there are error states for which it is not possible to write a good diagnostic message. We work around this issue via the static and dynamic techniques mentioned earlier (§4), namely (1) selective duplication of nonterminal symbols and (2) introduction of on `%on_error_reduce` declarations. Thus, we modify the grammar and the automaton, without affecting the language that is accepted. The pre-parser is described in the file `cparser/pre_parser.mly`.

After fine-tuning, the grammar has 138 nonterminal symbols, 94 terminal symbols, and 353 productions. It gives rise to a 597-state automaton. Our `%on_error_reduce` declarations cause reductions to be added in 101 states, thus preventing the detection of an error in any of these states. The reachability algorithm, applied to this automaton, gathers 2.4 million facts and 1.5 million edge facts in approximately 29 seconds, using 550Mb of memory. It reports that an error can occur in 212 states.

We manually inspect each of these states and write a collection of 157 distinct diagnostic message templates, which is stored in the file `cparser/handcrafted.messages`. Menhir compiles it to an OCaml function that maps a state number to a message template. This function is called from the file `cparser/ErrorReports.ml`, where we construct and display a full-fledged diagnostic message.

We allow a message template to contain the special form `$i`, where `i` is a literal integer index into the parser's stack. We replace this special form with the source text fragment that corresponds to this stack entry. This feature is not built into Menhir; it is implemented in CompCert using Menhir's stack inspection API. It helps explain how the input up to the error was interpreted.

For instance, in `index.c` (Figure 7), on line 3, a closing bracket is missing after the first occurrence of the array index `i`. This error is detected only upon reaching the semicolon. It is detected in a state which contains the LR(0) item `postfix_expression` → `postfix_expression LBRACK expression . RBRACK` among others. This implies that an opening bracket and a well-formed expression have been recognized, and that a closing bracket is a valid continuation (although not the only one). The expression that has just been recognized is currently the topmost entry in the parser's stack. Thus, we may use the special form `$0` in the message template: it is replaced at runtime with the underlying expression, namely `'i = b[i] = 0'`. In this example, although an error is

```
int convert (float c) {
  return (int) (c * 255.0;
}

$ ccomp -c missing.c
missing.c:2:26: syntax error after '255.0' and before ';'.
Ill-formed expression.
Up to this point, an expression has been recognized:
  'c * 255.0'
If this expression is complete,
then at this point, a closing parenthesis ')' is expected.
```

```
$ gcc -c missing.c
missing.c: In function 'convert':
missing.c:2:26: error: expected ')' before ';' token
   return (int) (c * 255.0;
                          ^
$ clang -c missing.c
missing.c:2:26: error: expected ')'
  return (int) (c * 255.0;
                         ^
missing.c:2:16: note: to match this '('
  return (int) (c * 255.0;
               ^
```

```
void f (void) {
  int a[50], b[50], i = 0;
  a[i = b[i] = 0;
}

$ ccomp -c index.c
index.c:4:19: syntax error after '0' and before ';'.
Ill-formed expression.
Up to this point, an expression has been recognized:
  'i = b[i] = 0'
If this expression is complete,
then at this point, a closing bracket ']' is expected.
```

```
$ gcc -c index.c
index.c: In function 'f':
index.c:3:17: error: expected ']' before ';' token
   a[i = b[i] = 0;
                ^
$ clang -c index.c
clang -c index.c
index.c:3:17: error: expected ']'
  a[i = b[i] = 0;
               ^
index.c:3:4: note: to match this '['
  a[i = b[i] = 0;
   ^
```

```
void f (void) { return; }|

$ ccomp -c bar.c
bar.c:1:26: syntax error after '}' and before '|'.
At this point, one of the following is expected:
  a function definition; or
  a declaration; or
  a pragma; or
  the end of the file.
```

```
$ gcc -c bar.c
bar.c:1:26: error: expected identifier or '('
    before '|' token
 void f (void) { return; }|
                          ^

$ clang -c bar.c
bar.c:1:26: error: expected identifier or '('
void f (void) { return; }|
                         ^
```

```
void f (void) { return; }}

$ ccomp -c braces.c
braces.c:1:26: syntax error after '}' and before '}'.
At this point, one of the following is expected:
  a function definition; or
  a declaration; or
  a pragma; or
  the end of the file.
```

```
$ gcc -c braces.c
braces.c:1:26: error: expected identifier or '('
    before '}' token
 void f (void) { return; }}
                          ^

$ clang -c braces.c
braces.c:1:26: error: extraneous closing brace ('}')
void f (void) { return; }}
                         ^
```

```
int f(void) { int x;) }

$ ccomp -c extra.c
extra.c:1:21: syntax error after ';' and before ')'.
At this point, one of the following is expected:
  a declaration; or
  a statement; or
  a pragma; or
  a closing brace '}'.
```

```
$ gcc -c extra.c
extra.c: In function 'f':
extra.c:1:21: error: expected statement before ')' token
 int f(void) { int x;) }
                     ^

$ clang -c extra.c
extra.c:1:21: error: expected expression
int f(void) { int x;) }
                    ^
```

```
int convert (float c) {
  return (int (c * 255.0));
}


$ ccomp -c render.c
render.c:2:16: syntax error after '(' and before 'c'.
Ill-formed declaration.
The following identifier is used as a type,
but has not been defined as such:
  'c'
```

```
$ gcc -c render.c
render.c: In function 'convert':
render.c:2:16: error: expected ')' before 'c'
   return (int (c * 255.0));
                ^

$ clang -c render.c
render.c:2:16: error: expected ')'
  return (int (c * 255.0));
               ^
render.c:2:15: note: to match this '('
  return (int (c * 255.0));
              ^
```

Figure 7. Examples of diagnostic messages produced by CompCert, `gcc` 5.3.0, and `clang` 3.6.0. (For the latter two, only one error shown.)

detected past the location of the actual mistake, the fact that the parser shows how it has interpreted the recent past suffices for the user to quickly locate and fix the mistake. `clang` produces a message of similar quality, whereas `gcc` does not say where the matching opening bracket is found.

In `bar.c` (Figure 7), an error is detected at the top level. We choose to explicitly list the possible futures at this point. `clang` and `gcc` suggest two terminal symbols as possible futures. This is low-level and incomplete, resulting (in our opinion) in a distinctly inferior diagnostic message. The next example, `braces.c`, differs only in the nature of the erroneous token, which is now a closing brace. This does not affect CompCert or `gcc`. `clang` recognizes this special case and reports the second closing brace as "extraneous". This is a situation where allowing the diagnostic message to depend on the erroneous symbol seems useful.

In `extra.c` (Figure 7), an error is detected inside the body of a function. Again, we explicitly list the possible futures at this point. `clang` and `gcc` both produce a correct yet incomplete list of futures: `gcc` requests a statement, whereas `clang` requests an expression, which is one particular kind of statement.

In `render.c` (Figure 7), one closing parenthesis is misplaced. The error is detected only upon looking ahead at the identifier `c`. (Indeed, "`int (`" appears to be the beginning of a function type.) In CompCert, due to the manner in which the "lexer hack" is set up, this error is detected in a state where a type name is the only valid future, yet one can tell that the parser has just looked ahead at a variable name. Our diagnostic message places emphasis on this problem. Furthermore, we indicate that we think we are engaged in a (parameter) declaration, which (with luck!) might help the reader understand that we have interpreted "`int (`" as the beginning of a function type. In comparison, `clang` and `gcc` request a closing parenthesis. This is correct, but incomplete (a parameter declaration would be a valid future), and also does not clearly indicate that the parser is trying to recognize a function type.

6. Related Work

Determining whether an error can occur in state s with lookahead symbol z amounts to determining whether the error entry at (s, z) in the action table is "inessential". Aho and Ullman [1] prove that, in a canonical LR(1) automaton, this is the case if and only if s is the target of a transition labeled with a non-terminal symbol A. In an SLR(1) automaton, one must add the condition that z is not in $\mathrm{FOLLOW}(A)$. Soisalon-Soininen [16] gives an algorithm for determining which error entries are inessential in *all* deterministic LR parsers of a grammar. These papers assume that the grammar is in the class LR(1). We do not make this hypothesis. We tolerate conflicts, and study reachability in the deterministic automaton obtained after conflict resolution. (In such an automaton, some states may be entirely unreachable.) To the best of our knowledge, this problem has not been previously studied.

Reachability (and, more generally, model-checking of temporal logics) in pushdown systems has been studied by several authors, including Bouajjani *et al.* [2]. A pushdown system is a pushdown automaton without an input. Bouajjani *et al.*'s algorithm for computing pre^* seems to bear some resemblance to our reachability algorithm. In fact, as noted by a reviewer, the lookahead behavior of an LR(1) automaton can be simulated by a pushdown system: a buffer of size 0 or 1 can be encoded in the control state. In summary, in principle, Bouajjani *et al.*'s algorithm can be used to solve the LR(1) reachability problem in polynomial time. More work would be required in order to assess its complexity in this particular case and compare it to the ad hoc algorithm that we have developed.

Gupta and Nandivada [6] analyze LL grammars where the lexer has multiple "lexical states" and may transition from one state to another either on its own or upon request by the parser. They detect dead (unreachable) productions, or in other words, situations where the parser expects a token that the lexer will never produce because it is in an inappropriate state. In comparison, Menhir does not have a notion of lexical state. The user is free to write semantic actions that influence the lexer (this is known as a "lexer hack", and is used in CompCert), but Menhir does not know about it. As noted by Gupta and Nandivada, this may lead Menhir to think that a configuration is reachable, when in fact it is not.

Following Horning [7] as well as later authors, such as Grune and Jacobs [5], one may distinguish error detection, error diagnosis, error recovery, and error repair. Detection consists in stopping when a syntax error is encountered. Quite obviously, every parser is able to do this, since it must accept the language generated by the grammar and nothing more. (However, not all parsing algorithms have the correct-prefix property, that is, the property of detecting an error as early as possible in a left-to-right scan of the input. LL and LR parsers have this property.) Diagnosis consists in detecting the earliest syntax error and explaining it, or more accurately, explaining the state of the parser at the point where the error is detected. Recovery consists in continuing beyond the first error, so as to be able to detect multiple errors, without attempting to deliver a parse tree. Repair consists in somehow fixing the input so as to continue beyond multiple errors and deliver a parse tree in the end.

We are interested in error diagnosis, which, as pointed out by Grune and Jacobs, seems to have received surprisingly little attention in the literature.

One relatively simple approach to error diagnosis is to compute and display the "acceptable set", that is, the set of terminal symbols that would have been valid at the error site. In recursive descent parsers, this idea has been studied by Wirth [18], Topor [17], and Kantorowitz and Laor [11], among others. The acceptable set can be used not just for diagnosis, but also for recovery. The parser generators ELL [4] and ANTLR [13] exploit this idea. In an LR parser produced by Menhir, computing the acceptable set at runtime would be straightforward, too. Indeed, provided the semantic actions are side-effect-free, the parsers generated by Menhir support taking a checkpoint and restarting from a checkpoint. So, the acceptable set can be computed by feeding the parser every terminal symbol and testing which ones are accepted. That said, in our opinion, printing the acceptable set as part of a diagnostic message is not a good idea (except perhaps when it is a singleton set). This set may contain many elements. For instance, in the CompCert C grammar, an expression can begin with one of 15 different terminal symbols; a statement can begin with one of 29 different symbols. This results in a low-level, overwhelming diagnostic message. It is desirable to describe the future at a higher level ("a statement is expected") and possibly beyond one symbol ("an expression, followed with a closing parenthesis, is expected").

The LALR parser generator `yacc` and its descendants, including `bison` [3] and Menhir itself, offer a special terminal symbol `error`, which can be used to program error diagnosis and error recovery strategies. When an error is detected, the generated parser first discards states from the stack until it gets back to a state in which the `error` symbol is acceptable. It consumes this symbol, entering a new state. (In a Menhir-generated parser, this may involve zero or more reductions, followed with shifting.) Then, it discards input symbols until it finds one that is acceptable in this state. At this point, it resumes parsing.

This mechanism can be used for error diagnosis. One extends the grammar with productions whose right-hand side ends in an `error` symbol and whose semantic action displays a diagnostic message and aborts. Used in this way, this mechanism appears to bear some similarity with `%on_error_reduce`, insofar as both mechanisms give access to dynamic information: that is, they allow the diagnostic message to depend on the contents of the stack.

However, the `error` symbol is notoriously difficult to use. It is hard to understand under which circumstances an `error` production is reduced, and (therefore) what the message should say. Furthermore, lacking tool support, it is impossible to ascertain that enough `error` productions have been added to cover all error situations. Finally, adding productions to the grammar could cause new conflicts to appear; one must be careful to avoid them.

Jeffery develops `merr` [8, 9] and equips Unicon with a set of diagnostic messages. The grammar (`uni/parser/unigram.y`) gives rise to a 453-state automaton, for which 70 messages are set up, covering 60 states (`uni/unicon/meta.err`). Our reachability algorithm reveals that the automaton has 224 states where an error can occur. Similarly, Pippijn van Steenhoven re-implements `merr` for (a slightly modified) Menhir and uses it to develop a C parser. For a 1031-state automaton, he sets up a collection of 70 diagnostic messages, which covers 55 states. Our algorithm reveals that the automaton has 391 states where an error can occur. These figures suggest that there is little hope of achieving complete coverage without tool support.

We improve upon Jeffery's technique by introducing an LR(1) reachability algorithm, which allows Menhir to show in which states an error may occur and how these states may be reached. Furthermore, as the grammar evolves, Menhir checks that the set of diagnostic messages remains correct, irredundant and complete We draw attention to the danger of over- or under-approximating the set of valid futures, and propose two ways of avoiding this danger, namely `%on_error_reduce` and selective duplication. We point out that adapting the grammar and automaton in such a way requires expertise.

Jeffery [9] writes: "In the case of Unicon, the use of Merr eliminated the initial, complete interpretation of some 360 states that was needed, as well as the subsequent reexamination of those states every time the grammar changes". Though Menhir automatically proposes a collection of input sentences that covers all error states, it is of course necessary to examine each of the error states so as to write a diagnostic message. Furthermore, when the grammar evolves, though Menhir checks that the collection of diagnostic messages remains correct, irredundant, and complete, there remains a small risk that one of the messages becomes inappropriate. (Perhaps it was associated with a sentence that led to a specific state, so the diagnostic message was worded in a specific way; but this sentence now leads to a state that can be reached in new ways, so the message must now be more general.) Thus, it remains in principle necessary to re-examine each of the error states after changing the grammar.

7. Conclusion

We have developed an algorithm for the reachability problem in LR(1) automata, implemented this algorithm in the Menhir parser generator, and used it to equip CompCert's ISO C99 parser with a collection of diagnostic messages that covers all of the states where an error can be detected.

The algorithm has been invaluable in understanding the landscape of the error states. Without it, we would have had to blindly write diagnostic messages for a set of supposedly common errors, and we would have been unable to see that, by modifying the grammar and automaton slightly, one can facilitate error diagnosis. Thanks to it, not only do we achieve completeness, but also we gain the ability to immediately evaluate how a modification of the grammar or automaton affects the set of error states. This has allowed us to better understand the differences between canonical and noncanonical automata, as well as the role of spurious reductions and the technique of selective duplication. This has led us to propose and implement `%on_error_reduce`, a mechanism for introducing more spurious reductions, therefore better controlling where errors are detected.

Our implementation of the reachability algorithm has been heavily optimized and can be used to analyze many real-world grammars, including CompCert's. Nevertheless, this algorithm is fundamentally quite expensive. It would be desirable to find an asymptotically more efficient algorithm, if one exists.

Acknowledgments

Gabriel Scherer provided motivation for this work. Jacques-Henri Jourdan explained several subtle aspects of the CompCert pre-parser. Arthur Charguéraud, Jean-Christophe Filliâtre, Jacques-Henri Jourdan, Yann Régis-Gianas, Gabriel Scherer proof-read the paper. The anonymous reviewers offered insightful remarks and pointers to the literature.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. A technique for speeding up LR(k) parsers. *SIAM Journal on Computing*, 2(2):106–127, 1973.

[2] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory (CONCUR)*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[3] Charles Donnelly and Richard Stallman. *Bison*, 2015.

[4] Josef Grosch. Efficient and comfortable error recovery in recursive descent parsers. *Structured Programming*, 11(3):129–140, 1990.

[5] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques: a practical guide, second edition*. Monographs in computer science. Springer, 2008.

[6] Kartik Gupta and V. Krishna Nandivada. Lexical state analyzer for JavaCC grammars. *Software: Practice and Experience*, 2015.

[7] James J. Horning. What the compiler should tell the user. In *Compiler Construction (CC)*, volume 21 of *Lecture Notes in Computer Science*, pages 525–548. Springer, 1974.

[8] Clinton L. Jeffery. *Merr User's Guide*, 2002.

[9] Clinton L. Jeffery. Generating LR syntax error messages from examples. *ACM Transactions on Programming Languages and Systems*, 25(5):631–640, 2003.

[10] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating $LR(1)$ parsers. In *European Symposium on Programming (ESOP)*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2012.

[11] E. Kantorowitz and H. Laor. Automatic generation of useful syntax error messages. *Software: Practice and Experience*, 16(7):627–640, 1986.

[12] Xavier Leroy. The CompCert C compiler. `http://compcert.inria.fr/`, 2015.

[13] Terence Parr. *The Definitive ANTLR 4 Reference, 2nd edition*. Pragmatic Bookshelf, 2013.

[14] François Pottier and Yann Régis-Gianas. The Menhir parser generator. `http://gallium.inria.fr/~fpottier/menhir/`.

[15] Paul Purdom. The size of LALR(1) parsers. *BIT Numerical Mathematics*, 14(3):326–337, 1974.

[16] Eljas Soisalon-Soininen. Inessential error entries and their use in LR parser optimization. *ACM Transactions on Programming Languages and Systems*, 4(2):179–195, 1982.

[17] Rodney W. Topor. A note on error recovery in recursive descent parsers. *ACM SIGPLAN Notices*, 17(2):37–40, 1982.

[18] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, 1978.