HAL archives-ouvertes.fr

Reachability and error diagnosis in LR(1) automata

François Pottier

▶ To cite this version:

François Pottier. Reachability and error diagnosis in LR(1) automata. Journées Francophones des Langages Applicatifs, Jan 2016, Saint-Malo, France. 2016, <http://jfla.inria.fr/2016/>. <hal-01248101>

HAL Id: hal-01248101
https://hal.inria.fr/hal-01248101

Submitted on 23 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reachability and error diagnosis in LR(1) automata

François Pottier

INRIA

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. Such an algorithm allows building a collection of pairs of an erroneous input sentence and a 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.

1. Introduction

LR parsers are powerful and efficient, but often do a poor job of explaining syntax errors. Although it is easy to report where an error was detected, it is 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 possible futures is encoded partly in the current state of the automaton and partly in its stack. We refer to the former part as *static* and to the latter part as *dynamic*. When constructing an error message, static information is readily available and seems easy to exploit, whereas dynamic information seems more difficult to exploit, since it is not obvious how to walk the stack and summarize its contents.

It may seem tempting to construct an error message based purely on static information, that is, based solely on the state in which the error is detected, regardless of the contents of the stack. Jeffery [3] describes such an approach. He suggests setting up a collection of erroneous input sentences, each of which is accompanied with a diagnostic message. From this data, his tool, `merr` [2], automatically produces a mapping of states to diagnostic messages[1].

In this paper, we wish to evaluate and improve the practicality of Jeffery's technique. In particular, we address the problem of building a collection of erroneous input sentences and diagnostic messages, and maintaining this collection as the grammar evolves.

Ideally, such a collection should 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. In fact, `merr` is independent of the parser generator, and does not even have access to the grammar or to a description of the automaton. All it requires is a way of running the generated parser and finding out in what state it fails. Jeffery advocates manually "growing" the collection of inputs and messages. One imagines that the collection

[1] Jeffery further argues in favor of taking into account not only the current state, but also the next input symbol. However, in his Unicon parser, he restricts his attention to just the current state. We follow this simpler approach.

is initially established by an expert, who studies the grammar and the automaton, and grown over time as end users report erroneous inputs that are not covered by the collection.

This state of affairs seems unsatisfactory. Tool support for achieving completeness initially, and for maintaining completeness as the grammar evolves, seems highly desirable. What is needed is the ability to produce a complete set of error states, together with input sentences that cause an error in each of these states.

This raises the problem of reachability in an LR(1) automaton. For every state s, is there an input sentence w that leads the automaton to the state s and causes an error to be detected in that state? After an informal look at the problem (§2), we propose an algorithm that answers this question, and produces a minimal witness sentence, when one exists (§3).

Equipped with this algorithm, which we have implemented in the Menhir parser generator [6], we handcraft a complete collection of diagnostic messages for the CompCert ISO C99 parser [5]. We draw several valuable 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 purely static approach, which does not seem to have been previously pointed out. In an LR(1) automaton, there may exist 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 don't know 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 describe two ways of working around it: by transforming the grammar, or by adding reduction actions to the automaton. We exploit both in the CompCert C parser (§5) and successfully work around the problem: our diagnostic messages never over-approximate the set of valid futures. So, 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 argue) is tolerable, and in fact profitable, in many situations. For instance, in the following C statement, after reading an incomplete expression, the parser encounters an invalid token, a semicolon:

```
color->y = (sc.kd * amb->y + il.y + sc.ks * is.y * sc.y;
```

Here, 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. Here is our diagnostic message:

```
render.c:70:57: syntax error after 'y' and before ';'.
Up to this point, a parenthesis '(' and an expression have been recognized:
  '(' 'sc.kd * amb->y + il.y + sc.ks * is.y * sc.y'
If this expression is complete,
then at this point, a closing parenthesis ')' is expected.
```

The hypothetical form "If this expression is complete, then ..." is our conventional way of pointing out that the list of futures proposed in the message is incomplete.

```
(* 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 {}
```

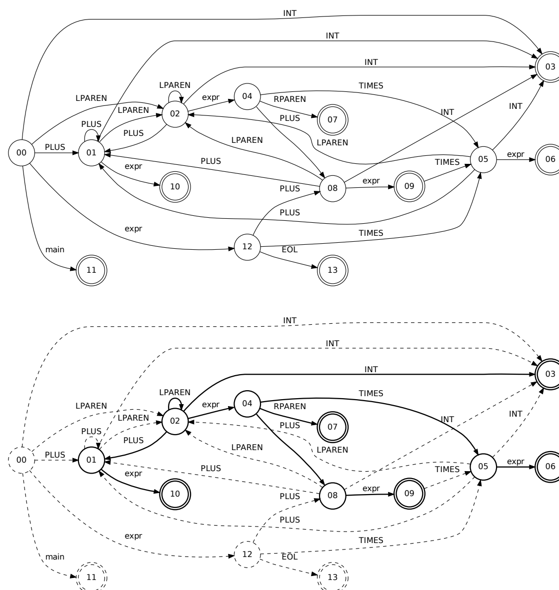

Figure 1: A grammar, its LR(1) automaton, and the star rooted at state 02 (§2.3).

2. The LR(1) reachability problem

In an ordinary finite-state automaton, the reachability problem is simple. It is a standard graph reachability problem in the automaton's state diagram and can be solved, say, by breadth-first search.

An LR(1) automaton, too, gives rise to a state diagram. Figure 1 offers an example[2]. There, one can still use breadth-first search to compute (shortest) paths towards every state. If no path towards a certain state s exists, one concludes that the automaton can never reach the state s. If such a path does exist, however, the situation is less clear-cut. An edge in this diagram is labeled either with a terminal symbol $z \in \Sigma$ or with a nonterminal symbol $A \in N$. The labels found along a path towards s form a sentential form $\alpha \in (\Sigma \cup N)^\star$. One may wonder whether the following "ideal property" holds:

> For every sentence in $w \in \mathcal{L}(\alpha)$,
> by consuming w, the automaton moves from its initial state to the state s.

If this property did hold, then one could conclude (assuming $\mathcal{L}(\alpha)$ is nonempty) that the automaton can reach the state s. Furthermore, as it is easy to compute an element of $\mathcal{L}(\alpha)$ whose length is minimal, one could exhibit a minimal sentence $w \in \Sigma^\star$ that takes the automaton to the state s.

We believe that this property holds if the grammar is in the class LR(0). In general, unfortunately, it does not hold. The reason is *lookahead*. The behavior of an LR(1) automaton depends not just on the sentence w that the automaton consumes, but also on the first input symbol z beyond w.

2.1. Why we must care about the next-to-last symbol

To illustrate this phenomenon, let us use the grammar in Figure 2. It is in the class LR(1). Its nonterminal symbols are `p`, `d`, and `phrase`. They generate the regular languages `P?`, `D*`, and

[2]The grammar is in the syntax of Menhir [6]. The braces `{}` denote empty semantic actions. This grammar is ambiguous; precedence declarations are used to "solve" conflicts. In the state diagram, at top right, the states where a reduction is permitted are doubly circled. The diagram does not show under which lookahead hypothesis each reduction is permitted, so it would be more accurate to say that this is a diagram of the underlying LR(0) automaton.

```
%token P D EOF
%start<unit> phrase
%%
phrase: p d EOF    {}
d     :   {} | d D {}
p     : P {} |     {}
```

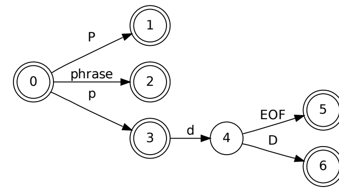

Figure 2: A grammar and its LR(1) automaton.

`P? D* EOF`, respectively. In the automaton, also shown in Figure 2, the initial state 0 has an edge labeled `p` to state 3, which has an edge labeled `d` to state 4, which rejects the terminal symbol `P`.

Let us ask two (related) questions about this automaton. (1) What is a minimal input sentence that, once consumed, takes the automaton to state 4? (2) What is a minimal input sentence that causes an error to be detected in state 4?

Based on the (incorrect) assumption that the "ideal property" holds, one might reason as follows. The nonterminal symbols `p` and `d` are nullable: they generate the empty sentence ϵ. Thus, one might think that, while consuming zero input symbols, the automaton can (and will, independently of the first input symbol) transition from state 0 through state 3 to state 4. Thus, one might conclude that the sentence ϵ is the answer to question (1). Furthermore, in state 4, if the next input symbol is `P`, the automaton fails. One might conclude that the sentence `P` is the answer to question (2).

This is incorrect, of course. The sentence `P` is a prefix of the valid sentence `P EOF`, so it is clear, without even inspecting the automaton, that it cannot cause an error. (Indeed, by construction, the automaton accepts every sentence that is valid as per the grammar.)

In reality, question (1) is ill-formulated. We should have asked: (1') what is a minimal input sentence w such that, by consuming w and looking ahead at z, the automaton reaches state 4?

In this example, the behavior of the automaton upon consuming the empty sentence ϵ depends on the first input symbol, as follows:

- If the first input symbol is `P`, then the automaton shifts: it consumes this input symbol and goes to state 1. There, it reduces `P` to `p`: it goes back to state 0, and from there, advances to state 3.
- If the first input symbol is not `P`, then the automaton reduces ϵ to `p`: it advances to state 3 without consuming this input symbol.

Either way, once in state 3, the automaton reduces ϵ to `d`. (This is a *default reduction*: it is carried out regardless of the lookahead symbol at this point.) This brings the automaton to state 4.

Thus, a correct answer to question (1') could be: "the sentence ϵ, if the next input symbol is not `P`; or the sentence `P`, regardless of the next input symbol".

Question (2), on the other hand, is well-formulated. It asks for a sentence wz such that consuming w takes the automaton to state 4 and, there, looking ahead at z causes the automaton to fail.

What is a correct answer to question (2)? We have noted above that the sentence `P` does not cause an error, as it is a prefix of a valid input. The same remark holds of the sentences `D` and `EOF`. Thus, no sentence of length 1 causes an error. On the other hand, it is not difficult to verify that the sentences `P P` and `D P` both lead to the detection of an error in state 4. Hence, either of them is a correct answer to question (2).

This example shows that, when in an LR automaton 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?".

2.2. Why we must care about the first symbol

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

2.3. 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 $A \rightarrow \alpha$ 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 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) $A \rightarrow \alpha$ 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, the bottom right part of Figure 1 shows the star rooted at state 02. Vertices and edges in bold are part of the star, while 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 1, 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.

2.4. Why we must tolerate non-canonical automata

A practical parser generator does not usually produce 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 point (3) removes shift actions and reduction actions (possibly making certain states entirely unreachable). Our reachability algorithm tolerates these alterations. It accepts any LR(1) automaton that is sound, not necessarily complete, with respect to the grammar.

$$\textsc{Init}\quad \dfrac{}{s \xrightarrow{\epsilon\,/\,\epsilon}\!\!\!\!\rightarrow s\,[z]}$$

$$\textsc{Step-Terminal}\quad \dfrac{s \xrightarrow{\alpha\,/\,w}\!\!\!\!\rightarrow s'\,[z] \qquad \mathcal{A} \vdash s' \xrightarrow{z} s''}{s \xrightarrow{\alpha z\,/\,wz}\!\!\!\!\rightarrow s''\,[z']}$$

$$\textsc{Step-NonTerminal}\quad \dfrac{\begin{array}{cc} s \xrightarrow{\alpha\,/\,w}\!\!\!\!\rightarrow s'\,[z] & \mathcal{A} \vdash s' \xrightarrow{A} s'' \\ s' \xrightarrow{A\,/\,w'} s''\,[z'] & z = \mathrm{first}(w'z') \end{array}}{s \xrightarrow{\alpha A\,/\,ww'}\!\!\!\!\rightarrow s''\,[z']}$$

$$\textsc{Reduce}\quad \dfrac{\begin{array}{cc} \mathcal{A} \vdash s \xrightarrow{A} s'' & s \xrightarrow{\alpha\,/\,w}\!\!\!\!\rightarrow s'\,[z] \\ \multicolumn{2}{c}{\mathcal{A} \vdash s' \text{ reduces } A \to \alpha \text{ on } z} \end{array}}{s \xrightarrow{A\,/\,w} s''\,[z]}$$

Figure 3: Inductive characterization of the predicates $s \xrightarrow{\alpha\,/\,w}\!\!\!\!\rightarrow s'\,[z]$ and $s \xrightarrow{A\,/\,w} s'\,[z]$.

3. An LR(1) reachability algorithm

All our algorithm needs to know about the automaton is which transitions exist and which reductions are permitted. 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. We write "$\mathcal{A} \vdash s$ reduces $A \to \alpha$ 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 do not recall the small-step dynamic semantics of LR automata. In short, the automaton maintains a stack of states; the topmost state on the stack is the "current" state. When a (terminal or nonterminal) transition is taken, the new state is pushed onto the stack. When a production $A \to \alpha$ is reduced, $|\alpha|$ states are popped off the stack, which means that the automaton restores the stack that existed before recognizing α. 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 (§2.3). Such a fact should be informally interpreted as follows:

> 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, ending up in state s',
> and consumes w.

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 3. 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 $\text{first}(w'z')$ 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 this paper, we take it for granted that this big-step dynamic semantics is correct and complete with respect to the small-step dynamic semantics. In a more ambitious treatment, one would prove the two semantics equivalent.

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:

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

(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) and the same first input symbol (namely, $\text{first}(w_1 z)$, also known as $\text{first}(w_2 z)$). 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 3, 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 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 (§2.3). 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 4. It follows the rules of Figure 3 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 $n.|\mathcal{G}|$ (§2.3).

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.

Because priorities are low nonnegative integers, a simple-minded, array-based priority queue can be

procedure STEP-TERMINAL($s \xrightarrow{\alpha\,/\,w} s'\,[z]$)
  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 $s \xrightarrow{\alpha z\,/\,wz} s''\,[z']$ **into** Q

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

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

procedure REDUCE($s \xrightarrow{\alpha\,/\,w} s'\,[z]$)
  if $\mathcal{A} \vdash s'$ reduces $A \to \alpha$ on z **then** — for some non-terminal symbol A, that is
    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 $s \xrightarrow{\epsilon\,/\,\epsilon} s\,[z]$ **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 4: The reachability algorithm.

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.

The set of facts F must support the following operations.

1. Test whether a fact $s \xrightarrow{\alpha\,/\,w} s'\,[z]$ 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 $s \xrightarrow{\alpha\,/\,w} s'\,[z]$. 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 $z = \text{first}(wz')$. This is used in procedure Step-NonTerminal-Left.

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

3.4. 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. (Details omitted.) 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.

3.5. Performance aspects

We ran the reachability algorithm (including the postprocessing phase of §3.4) on a 40-core Intel Xeon running at 2.4GHz, equipped with 1Tb of RAM[3]. We tried 200 grammars found in Menhir's test suite, using LALR as the construction method. Of these, 88 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 5. 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 [7].

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

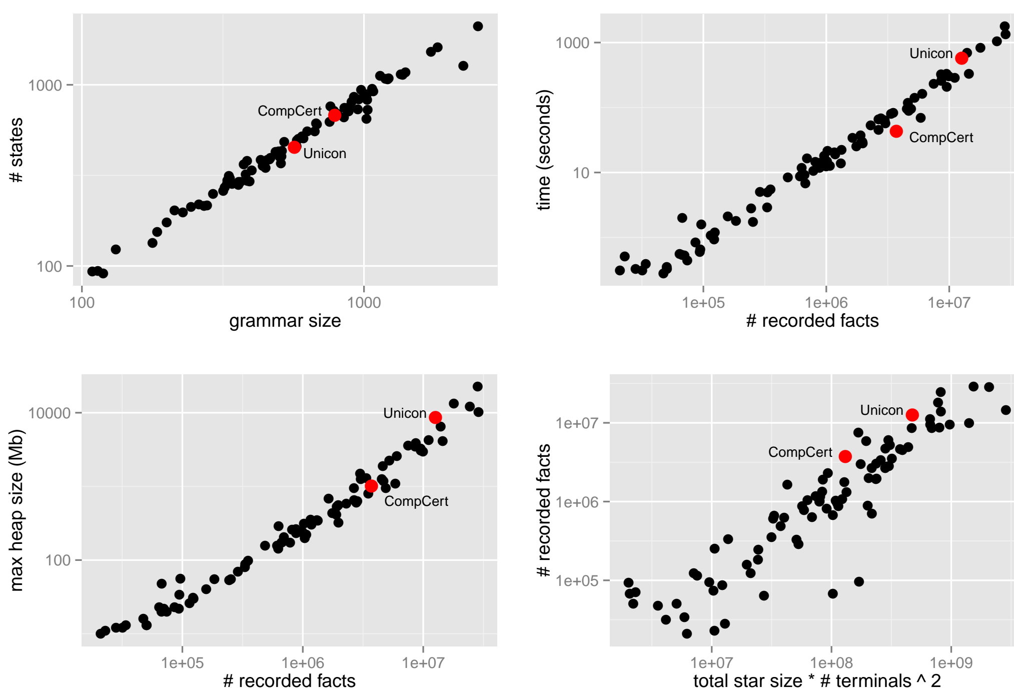

Figure 5: Performance aspects of the reachability algorithm.

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 1.22: the running time grows super-linearly with the number of facts.

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 1.03: the correlation seems linear. The estimated slope in the original diagram is 5.10^{-4}, that is, about 500Mb per million facts. For Unicon, (a newer version of) the grammar studied by Jeffery [3], 12 million facts are recorded, and the space usage is 8Gb.

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 13x as much time and leads to gathering 3.4x as many facts. Furthermore, the time required to process CompCert's grammar itself varies widely, depending on which version of the grammar one analyzes: 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 1.001, 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.

4. Designing 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` $\rightarrow$ `LBRACK expr RBRACK`. The LALR automaton for this extended grammar contains a state where reducing the production `expr` $\rightarrow$ `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:

$$\begin{array}{rcl} \texttt{expr} & \rightarrow & \texttt{LPAREN snl(COMMA, expr) RPAREN} \\ \texttt{expr} & \rightarrow & \texttt{LBRACK snl(COMMA, expr) RBRACK} \end{array}$$

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

$$\begin{array}{rcll} \texttt{expr} & \rightarrow & \texttt{expr} \bullet \texttt{PLUS expr} & [\ldots] \\ \texttt{expr} & \rightarrow & \texttt{expr} \bullet \texttt{TIMES expr} & [\ldots] \\ \texttt{snl(COMMA, expr)} & \rightarrow & \texttt{expr} \bullet & [\texttt{RPAREN}, \texttt{RBRACK}] \\ \texttt{snl(COMMA, expr)} & \rightarrow & \texttt{expr} \bullet \texttt{COMMA snl(COMMA, expr)} & [\ldots] \end{array}$$

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 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:

$$\texttt{expr} \rightarrow \texttt{LBRACK snl(COMMA, expr)} \bullet \texttt{RBRACK} \quad [\ldots]$$

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 therefore, 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

[4] Menhir's library defines a parameterized nonterminal symbol `separated_nonempty_list(sep, elem)`, which we abbreviate as `snl(sep, elem)`.

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 further details, see Menhir's documentation [6].

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 further details, see Menhir's documentation [6].

5. Application to CompCert C

We have extended Menhir [6] 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.

Using these tools, we have developed diagnostic messages for the CompCert C "pre-parser" [4, 5]. We found at first that, due to a lack of static information, it was not always possible to associate a good error message with every error state. We worked around this problem by combining the static and dynamic techniques mentioned earlier, namely (1) selective duplication of nonterminal symbols and (2) introduction of on `%on_error_reduce` declarations.

After fine-tuning, the grammar has 145 nonterminal symbols, 93 terminal symbols, and 365 productions. For this grammar, Menhir produces a 677-state automaton. We use `%on_error_reduce` to add reductions in 99 states, thus preventing the detection of an error in any of these states. The reachability algorithm gathers 3.7 million facts and 2.2 million edge facts in approximately 43 seconds, using 1Gb of memory. (In the absence of `%on_error_reduce` declarations, the algorithm gathers only 1.3 million facts in 13 seconds, using 0.3 gigabyte of heap space!) The algorithm reports that an error can occur in 263 states, for which we set up 150 distinct diagnostic messages.

We allow a diagnostic message to contain the special form `$i`, where `i` is an integer literal. This form is automatically replaced at runtime with the fragment of the input text that corresponds to the `i`-th stack cell. This helps explain how the input up to the error was interpreted, as in the following example. A closing bracket is missing after the first occurrence of the index `i`:

```
multvec_i[i = multvec_j[i] = 0;
```

CompCert produces the following diagnostic message:

```
subsumption.c:71:34: syntax error after '0' and before ';'.
Ill-formed expression.
Up to this point, an expression has been recognized:
  'i = multvec_j[i] = 0'
If this expression is complete,
```

```
then at this point, a closing bracket ']' is expected.
```

The message template contains `$0`, which is replaced at runtime with `'i = multvec_j[i] = 0'`. In this example, although a failure occurs past the location of the actual error, the fact that the parser shows how it has interpreted the recent past suffices for the user to quickly locate and fix the mistake.

6. Related work

To the best of our knowledge, reachability in LR automata 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.*[1]. 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. Although the two reachability problems are related, they differ in a few aspects: the manner in which the stack evolves is perhaps more restricted in an LR(1) automaton; on the other hand, the treatment of lookahead introduces extra complication.

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 complete collection of erroneous input sentences and diagnostic messages.

The algorithm has been invaluable in understanding the landscape of the states where an error can be detected. Without it, we would have had to blindly write diagnostic messages for a set of supposedly common errors. 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 states where an error can occur. 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 a mechanism for intentionally introducing spurious reductions, therefore better controlling where errors are detected.

References

[1] A. Bouajjani, J. Esparza, and O. 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.

[2] C. L. Jeffery. *Merr User's Guide*, 2002.

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

[4] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating *LR*(1) parsers. In *European Symposium on Programming (ESOP)*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2012.

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

[6] F. Pottier and Y. Régis-Gianas. The Menhir parser generator. `http://gallium.inria.fr/~fpottier/menhir/`.

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