Validating $LR(1)$ Parsers

Jacques-Henri Jourdan[1,2], François Pottier[2], and Xavier Leroy[2]

[1] École Normale Supérieure
[2] INRIA Paris-Rocquencourt

Abstract. An $LR(1)$ parser is a finite-state automaton, equipped with a stack, which uses a combination of its current state and one lookahead symbol in order to determine which action to perform next. We present a validator which, when applied to a context-free grammar $\mathcal{G}$ and an automaton $\mathcal{A}$, checks that $\mathcal{A}$ and $\mathcal{G}$ agree. Validating the parser provides the correctness guarantees required by verified compilers and other high-assurance software that involves parsing. The validation process is independent of which technique was used to construct $\mathcal{A}$. The validator is implemented and proved correct using the Coq proof assistant. As an application, we build a formally-verified parser for the C99 language.

1 Introduction

Parsing remains an essential component of compilers and other programs that input textual representations of structured data. Its theoretical foundations are well understood today, and mature technology, ranging from parser combinator libraries to sophisticated parser generators, is readily available to help implementing parsers.

The issue we focus on in this paper is that of *parser correctness*: how to obtain formal evidence that a parser is correct with respect to its specification? Here, following established practice, we choose to specify parsers via context-free grammars enriched with semantic actions.

One application area where the parser correctness issue naturally arises is formally-verified compilers such as the CompCert verified C compiler [14]. Indeed, in the current state of CompCert, the passes that have been formally verified start at abstract syntax trees (AST) for the CompCert C subset of C and extend to ASTs for three assembly languages. Upstream of these verified passes are lexing, parsing, type-checking and elaboration passes that are still in need of formal verification in order to attain end-to-end verification. The present paper addresses this need for the parsing pass. However, its results are more generally applicable to all high-assurance software systems where parsing is an issue.

There are many ways to build confidence in a parser. Perhaps the simplest way is to instrument an unverified parser so that it produces full parse trees, and, at every run of the compiler, check that the resulting parse tree conforms to the grammar. This approach is easy to implement but does not establish completeness (all valid inputs are accepted) or unambiguity (for each input, there is at most one parse tree). Another approach is to apply program proof

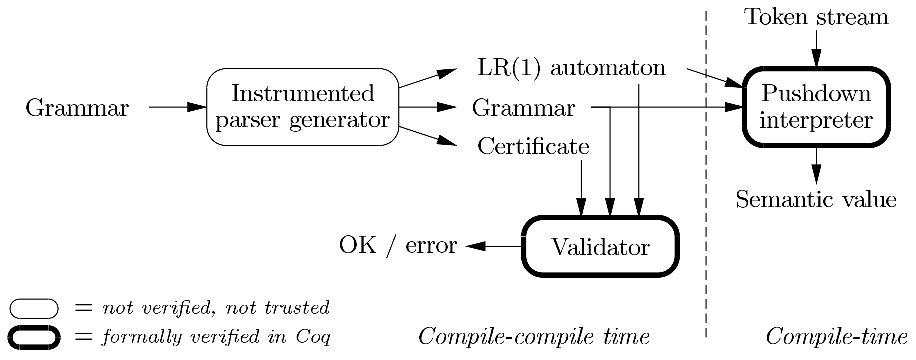

Fig. 1. General architecture.

directly to a hand-written or generated parser. However, such a proof is tedious, especially if the parser was automatically generated, and must be re-done every time the the parser is modified. Yet another approach, developed by Barthwal and Norrish [4,3], is to formally verify, once and for all, a parser generator, guaranteeing that whenever the verified generator succeeds, the parser that it produces is correct with respect to the input grammar. However, Barthwal and Norrish's proof is specific to a particular parser generator that only accepts *SLR* grammars. It so happens that the ISO C99 grammar we are interested in is not *SLR*. Before being applicable to CompCert, Barthwal and Norrish's work would, therefore, have to be extended in nontrivial ways to a richer class of grammars such as *LALR*.

In this paper, we develop a fourth approach: *a posteriori* verified validation of an $LR(1)$ automaton produced by an untrusted parser generator, as depicted in Fig. 1. After every run of the parser generator (that is, at *compile-compile time*), the source grammar $\mathcal{G}$, the generated automaton $\mathcal{A}$, and auxiliary information acting as a certificate are fed in a *validator*, which checks whether $\mathcal{A}$ recognizes the same language as $\mathcal{G}$. If so, the build of the compiler proceeds; if not, it is aborted with an error.

The first contribution of this paper is the algorithm that performs this validation. It is relatively simple, and, to the best of our knowledge, original. It applies indifferently to many flavors of $LR(1)$ automata, including $LR(0)$ (a degenerate case), *SLR* [5], *LALR* [2], Pager's method [17], and canonical $LR(1)$ [12]. The second contribution is a soundness proof for this algorithm, mechanized using the Coq proof assistant, guaranteeing with the highest confidence that if the validation of an automaton $\mathcal{A}$ against a grammar $\mathcal{G}$ succeeds, then the automaton $\mathcal{A}$ and the interpreter that executes it form a correct parser for $\mathcal{G}$. The last contribution is an experimental assessment of our approach over the ISO C99 grammar, demonstrating applicability to realistic parsers of respectable size.

In summary, the approach to high-confidence parsing developed in this paper is attractive for several reasons: (1) it provides correctness guarantees about an

$LR(1)$ parser as strong as those obtained by verifying a $LR(1)$ parser generator; (2) only the validator needs to be formally verified, but not the parser generator itself, reducing the overall proof effort; (3) the validator and its soundness proof are reusable with different parser generators and different flavors of $LR(1)$ parsing; (4) existing, mature parser generators such as Menhir [19] can be used with minimal instrumentation, giving users the benefits of the extensive diagnostics produced by these generators.

This paper is organized as follows. In §2, we review context-free grammars, $LR(1)$ automata, and their meaning. In §3, we establish three properties of automata, namely soundness (§3.1), safety (§3.2) and completeness (§3.3). Safety and completeness are true only of some automata: we present a set of conditions that are sufficient for these properties to hold and that can be automatically checked by a validator. After presenting some facts about our Coq implementation (§4), we discuss its application to a C99 parser in the setting of CompCert (§5). We conclude with discussions of related work (§6) and directions for future work (§7).

Our modifications to Menhir are available as part of the standard release [19]. Our Coq code is not yet part of the CompCert release, but is available online [10].

2 Grammars and Automata

2.1 Symbols

We fix an alphabet of *terminal symbols* a and an alphabet of *non-terminal symbols* A, where an *alphabet* is a finite set. A *symbol* X is either a terminal symbol or a non-terminal symbol. We write α for a *sentential form*, that is, a finite sequence of symbols.

2.2 Grammars

We fix a grammar $\mathcal{G}$, where a *grammar* consists of:

1. a *start symbol* S;
2. an alphabet of *productions* p;
3. for every symbol X, a type $[\![X]\!]$.

The first two components are standard. (The syntax of productions will be presented shortly.) The third component, which is not usually found in textbooks, arises because we are interested not in recognition, but in parsing: that is, we would like not only to decide whether the input is valid, but also to construct a *semantic value*. In other words, we would like to consider that a grammar defines not just a *language* (a set of words) but a *relation* between words and semantic values. However, before we do so, we must answer the question: what is the type of semantic values? It should be decided by the user, that is, it should be part of the grammar. Furthermore, one should allow distinct symbols to carry different types of semantic values. For instance, a terminal symbol that stands for an identifier might carry a string, while a non-terminal symbol that stands for an arithmetic expression might carry an abstract syntax tree, that is, a value of a user-defined inductive type. If we were to force the user to adopt a single universal type of semantic values, the user's code would become polluted with tags and dynamic tests, and it would not be possible for the user to argue that these tests cannot fail. For this reason, for every symbol X, we allow the user to choose the type $[\![X]\!]$ of the semantic values carried by X. (In Coq, $[\![X]\!]$ has type `Type`.) By abuse of notation, if α is a (possibly empty) sequence of symbols $X_1 \ldots X_n$, we write $[\![\alpha]\!]$ for the tuple type $[\![X_1]\!] \times \ldots \times [\![X_n]\!]$.

How are semantic values constructed? The answer is two-fold. The semantic values carried by terminal symbols are constructed by the lexer: in other words, they are part of the input that is submitted to the parser. The semantic values carried by non-terminal symbols are constructed by the parser: when a production is reduced, a semantic action is executed. Let us now examine each of these two aspects in greater detail.

The input of the parser is a stream of tokens, where a *token* is a dependent pair (a, v) of a terminal symbol a and a semantic value v of type $[\![a]\!]$. We assume that this stream is infinite. There is no loss of generality in this assumption: if one wishes to work with a finite input stream, one can complete it with an infinite number of copies of a new "end-of-stream" token. In the following, we write w for a finite sequence of tokens and ω for an infinite stream of tokens.

A *production* p is a triple of the form $A \longrightarrow \alpha\ \{f\}$. (Above, we have written that productions form an alphabet, that is, they are numbered. We abuse notation and elide the details of the mapping from productions-as-numbers to productions-as-triples.) The left-hand side of a production is a non-terminal symbol A. The right-hand side consists of a sentential form α and a *semantic action* f of type $[\![\alpha]\!] \rightarrow [\![A]\!]$. The semantic action, which is provided by the user, indicates how a tuple of semantic values for the right-hand side α can be transformed into a semantic value for the left-hand side A. In our approach, the semantic action plays a dual role: on the one hand, it is part of the grammar, and plays a role in the definition of the semantics of the grammar; on the other hand, it is used, at runtime, by the parser. The semantic action is a Coq function and is supplied by the user as part of the grammar. (The parser generator Menhir views semantic actions as pieces of text, so no modification is needed for it to support Coq semantic actions instead of Objective Caml semantic actions.)

2.3 Semantics of Grammars

The semantics of grammars is usually defined in terms of a relation between symbols X and words w, written $X \longrightarrow w$, pronounced: "X derives w". As announced earlier, we extend this relation with a third parameter, a semantic value v of type $[\![X]\!]$. Thus, we write $X \xrightarrow{v} w$, pronounced: "X derives w producing v". The inductive definition of this relation is as follows:

$$a \xrightarrow{v} (a, v) \qquad \frac{\begin{array}{c} A \longrightarrow X_1 \dots X_n \; \{f\} \text{ is a production} \\ \forall i \in \{1, \dots, n\} \qquad X_i \xrightarrow{v_i} w_i \end{array}}{A \overset{f(v_1,\dots,v_n)}{\longrightarrow} w_1 \dots w_n}$$

This semantics is used when we state that the parser is sound and complete with respect to the grammar (Theorems 1 and 3).

2.4 Automata

We fix an automaton $\mathcal{A}$, where an *automaton* consists of:

1. an alphabet of *states* σ, with a distinguished *initial state*, written *init*;
2. an action table;
3. a goto table;
4. for every non-initial state σ, an incoming symbol, written *incoming*(σ).

A *non-initial state* is a state other than *init*.

An *action table* is a total mapping of pairs (σ, a) to actions. The idea is, if σ is the current state of the automaton and a is the terminal symbol currently found at the head of the input stream, then the corresponding action instructs the automaton what to do next. An *action* is one of *shift* σ' (where σ' is non-initial), *reduce* p (where p is a production), *accept*, and *reject*. In particular, the situation where the action table maps (σ, a) to *shift* σ' can be thought of graphically as an edge, labeled a, from σ to σ'.

Remark 1. Our description of the action table as a mapping of pairs (σ, a) to actions appears to imply that the parser *must* peek at the next input token a *before* it can decide upon the next action. This might seem perfectly acceptable: because we have assumed that the input stream is infinite, there always is one more input token. In practice, however, more care is required. There are situations where the input stream is effectively infinite and nevertheless one must not allow the parser to systematically peek at the next token. For instance, the input stream might be connected to a keyboard, where a user is entering commands. If the parser has been asked to recognize one command, then, upon finding the end of a command, it should terminate and report success *without* attempting to read one more token: otherwise, it runs the risk of blocking until further keyboard input is available!

In order to address this problem, we allow our automata to sometimes take a *default action* without peeking at the next input token. We adopt the following convention: if, for a certain state σ and for all terminal symbols a, the entries found at (σ, a) in the action table are identical, then the automaton, when in state σ, determines which action should be taken *without* consulting the input stream[3].

[3] Of course, it would be inefficient to naïvely test whether one entire column of the action table contains identical entries. In reality, Menhir produces (and our Coq

A *goto table* is a partial mapping of pairs (σ, A) to non-initial states. If the goto table maps (σ, A) to σ', then the automaton can take a transition from σ to σ' after recognizing a word that derives from A. This can be thought of graphically as an edge, labeled A, from σ to σ'. This table is a partial mapping: a state σ may have no outgoing edge labeled A.

A well-formed LR automaton has the property that, for every non-initial state σ, all of the edges that enter σ carry a common label. (The initial state *init* has no incoming edges.) We refer to this label as the *incoming symbol* of σ. Although we could have our validator reconstruct this information, we ask the user to supply it as part of the description of the automaton. We require that this information be consistent with the action and goto tables, as follows. If the action table maps (σ, a) to *shift* σ', then we require $\mathit{incoming}(\sigma') = a$. Similarly, if the goto table maps (σ, A) to σ', then we require $\mathit{incoming}(\sigma') = A$. We encode these requirements directly in the types of the action and goto tables, using dependent types, so we do not need to write validation code for them.

2.5 Semantics of Automata

We give semantics to automata by defining an interpreter for automata. The interpreter is a function named $\mathit{parse}(\cdot, \cdot)$. Its first (and main) argument is a token stream ω. We need an auxiliary argument, the "fuel", which is discussed further on.

The interpreter maintains a *stack* s, which is a list of dependent pairs of a non-initial state σ and a semantic value v of type $[\![\mathit{incoming}(\sigma)]\!]$. Indeed, $\mathit{incoming}(\sigma)$ is the (terminal or non-terminal) symbol that was recognized prior to entering state σ, and v is a semantic value associated with this symbol. We write $s(\sigma, v)$ for a stack whose top cell is the pair (σ, v) and whose tail is the stack s. At the beginning of a run, the stack is empty. At every moment, the *current state* of the automaton is the state found in the top stack cell if the stack is non-empty; it is the initial state *init* if the stack is empty[4].

In several places, the interpreter can generate an internal error, revealing a flaw in the automaton. Indeed, it is sometimes much easier to write an interpreter that can encounter an internal error, and prove *a posteriori* that this situation never arises if the automaton satisfies certain properties, than to define *a priori* an interpreter than never encounters an internal error. In other words, instead of hardwiring *safety* (that is, the absence of internal errors) into the definition of the interpreter, we make it a separate theorem (Theorem 2).

We use an error monad to deal with internal errors. In this paper, we use $\lightning$ to denote an internal error. By abuse of notation, we elide the "return" operation

code uses) a two-level action table, where the first level is indexed only by a state σ and indicates whether there exists a default action, and the second level (which is consulted only if there is no default action) is indexed by a state σ and a terminal symbol a.

[4] In many textbooks, one does not consider semantic values, so the stack is a list of states; at the beginning of a run, the stack is a singleton list of the state *init*; the stack is never empty, so the current state is always the one found on top of the stack.

of the error monad. Thus, the interpreter produces either $\lightning$ or a parse result (defined below).

We also need a way of dealing with the possibility of non-termination. Again, it is not possible to prove *a priori* that the interpreter terminates. When an LR automaton reduces a unit production (of the form $A \longrightarrow A'$) or an epsilon production (of the form $A \longrightarrow \varepsilon$), the size of the stack does not decrease. There do exist automata, as per the definition of §2.4, whose interpretation does not terminate. It is not clear, at present, what property one could or should require of the automaton in order to ensure termination. (We discuss this issue in §7.)

We adopt a simple and pragmatic approach to this problem: in addition to the token stream, the interpreter requires an integer argument n, which we refer to as the *fuel*. In the main loop of the interpreter, each iteration consumes one unit of fuel. If the interpreter runs out of fuel, it stops and reports that it was not able to terminate normally. We write *out-of-fuel* for this outcome.

Thus, a *parse result* is one of: *out-of-fuel*, which was explained above; *reject*, which means that the input is invalid; and *parsed* v ω, which means that the input is valid, that the semantic value associated with the prefix of the input that was recognized is v, and that the remainder of the input is ω. (The value v has type $[\![S]\!]$, where S is the start symbol of the grammar. The value ω is a token stream.)

In summary, the interpreter accepts a token stream and a certain amount of fuel and produces either $\lightning$ or a parse result, as defined above.

The interpreter works in a standard manner. At each step, it looks up the action table at (σ, a), where σ is the current state of the automaton and (a, v) is the next input token. Then,

1. if the action is *shift* σ', the input token (a, v) is consumed, and the new cell (σ', v) is pushed onto the stack. Because $\mathit{incoming}(\sigma') = a$ holds, it is possible to *cast* the value v from the type $[\![a]\!]$ to the type $[\![\mathit{incoming}(\sigma')]\!]$: this is required for this new stack cell to be well-typed.
2. if the action is *reduce* $A \longrightarrow X_1 \dots X_n$ $\{f\}$, the interpreter *attempts* to pop n cells off the stack, say $(\sigma_1, v_1) \dots (\sigma_n, v_n)$, and *dynamically checks* that $\mathit{incoming}(\sigma_i) = X_i$ holds for every $i \in \{1, \dots, n\}$. If the stack does not have at least n cells, or if this check fails, then an internal error occurs. Otherwise, thanks to the success of these dynamic checks, each of the semantic values v_i can be cast from the type $[\![\mathit{incoming}(\sigma_i)]\!]$ to the type $[\![X_i]\!]$. Thus, the tuple $(v_1, \dots, v_n)$ admits the type $[\![X_1 \dots X_n]\!]$, and is a suitable argument for the semantic action f. The application of f to this tuple yields a new semantic value v of type $[\![A]\!]$. There remains for the interpreter to consult the goto table at the current state and at the non-terminal symbol A. If this entry in the goto table is undefined, an internal error occurs. Otherwise, this entry contains a state σ', and (after another cast) the new cell (σ', v) is pushed onto the stack.
3. if the action is *accept*, the interpreter attempts to pop one cell off the stack, say (σ, v), and checks that $\mathit{incoming}(\sigma) = S$ holds, where S is the start symbol of the grammar. Thus, the value v can be cast to the type $[\![S]\!]$.

(This can be thought of as reducing a special production $S' \longrightarrow S$.) The interpreter then checks that the stack is now empty and terminates with the parse result *parsed* $v\ \omega$, where ω is what remains of the input stream.
4. if the action is *reject*, the interpreter stops with the parse result *reject*.

In summary, there are four possible causes for an internal error: a dynamic check of the form $\mathit{incoming}(\sigma) = X$ may fail; an attempt to pop a cell off the stack fails if the stack is empty; an attempt to consult the goto table fails if the desired entry is undefined; an attempt to accept fails if the stack is nonempty.

3 Correctness Properties and Validation

We now show how to establish three properties of the automaton $\mathcal{A}$ with respect to the grammar $\mathcal{G}$. These properties are *soundness* (the parser accepts only valid inputs), *safety* (no internal error occurs), and *completeness* (the parser accepts all valid inputs). By design of our interpreter, the first property is true of all automata, whereas the latter two are true only of some automata. For safety and for completeness, we present a set of conditions that are sufficient for the desired property to hold and that can be automatically checked by a validator.

3.1 Soundness

The soundness theorem states that if the parser accepts a finite prefix w of the input stream ω, then (according to the grammar) the start symbol S derives w. More precisely, if the parser accepts w and produces a semantic value v, then v is the value associated with this particular derivation of w from S, that is, the relation $S \xrightarrow{v} w$ holds.

Theorem 1 (Soundness). *If* $\mathit{parse}(\omega, n) = \mathit{parsed}\ v\ \omega'$ *holds, then there exists a word* w *such that* $\omega = w\omega'$ *and* $S \xrightarrow{v} w$.

No hypotheses about the automaton are required, because the situations where "something is wrong" and soundness might be endangered are detected at runtime by the interpreter and lead to internal errors. In other words, we have shifted most of the burden of the proof from the soundness theorem to the safety theorem.

In order to prove this theorem, it is necessary to establish an invariant stating that the symbols associated with the states found in the stack derive the input word that has been consumed. For this purpose, we introduce a new predicate, written $s \Longrightarrow w$, which relates a stack s with a token word w. It is inductively defined as follows:

$$\varepsilon \Longrightarrow \varepsilon \qquad\qquad \frac{s \Longrightarrow w_1 \qquad \mathit{incoming}(\sigma) \xrightarrow{v} w_2}{s(\sigma, v) \Longrightarrow w_1 w_2}$$

Then, the main soundness invariant can be stated as follows: if the parser has consumed the input word w and if the current stack is s, then $s \Longrightarrow w$ holds.

3.2 Safety

The safety theorem states that if the automaton passes the *safety validator* (which we describe further on) then the interpreter never encounters an internal error. A *validator* is a Coq term of type `bool`, which has access to the grammar, to the automaton, and to certain additional annotations that serve as hints (and which we describe below as well). Thus, the safety theorem takes the following form:

Theorem 2 (Safety). *If the criteria enforced by the safety validator are satisfied, then* $parse(\omega, n) \neq \lightning$ *for every input stream* ω *and for every integer "fuel"* n.

All of the causes of internal errors that were previously listed (§2.5) have to do with a stack that does not have the expected shape (e.g., it has too few cells) or contents (e.g. some cell contains a semantic value of inappropriate type, or contains a state for which no entry exists in the goto table). Thus, in order to ensure safety, we must have precise control of the shape and contents of the stack.

Recall that a stack s is a sequence of pairs $(\sigma_1, v_1) \dots (\sigma_n, v_n)$. In what follows, it is convenient to abstractly describe the structure of such a stack in two ways. First, we are interested in the underlying sequence of symbols: we write $\mathit{symbols}(s)$ for the sequence of symbols $\mathit{incoming}(\sigma_1) \dots \mathit{incoming}(\sigma_n)$. Second, we are interested in the underlying sequence of states: we write $\mathit{states}(s)$ for the sequence of singleton sets $\{\mathit{init}\}\{\sigma_1\} \dots \{\sigma_n\}$. (We use singleton sets here because we will shortly be interested in approximating these singleton sets with larger sets of states Σ.)

A key remark is the following: the sequences $\mathit{symbols}(s)$ and $\mathit{states}(s)$ are not arbitrary. They follow certain patterns, or, in other words, they respect a certain invariant. This invariant will be sufficient to guarantee safety.

How do we find out what this invariant is? Two approaches come to mind: either the safety validator could reconstruct this invariant by performing a static analysis of the automaton (this would require a least fixed point computation), or the parser generator could produce a description of this invariant, which the safety validator would verify (this would require checking that a candidate fixed point is indeed a fixed point). We somewhat arbitrarily adopt the latter approach. The former approach appears viable as well, especially if one exploits a pre-existing verified library for computing least fixed points.

Thus, the annotations that the safety validator requires (and that the parser generator must produce) form a description of the safety invariant. For each non-initial state σ, these annotations are:

1. a sequence of symbols, written $\mathit{pastSymbols}(\sigma)$;
2. a sequence of sets of states, written $\mathit{pastStates}(\sigma)$.

(There is a redundancy, which we discuss in §7.) These annotations are meant to represent approximate (conservative) information about the shape and contents of the stack. In order to explain their meaning, let us now define the safety invariant in terms of these annotations.

We begin by defining the relations that exist between abstract descriptions of the stack and concrete stacks. We need two such relations. The *suffix ordering* between two *sequences of symbols* is defined in the usual way: that is, $X_m \dots X_1$ is a suffix of $X'_n \dots X'_1$ if and only if $m \leq n$ holds and $X_i = X'_i$ holds for every $i \in \{1, \dots, m\}$. The *suffix ordering* between two *sequences of sets of states* is defined in the same manner, up to pointwise superset ordering: that is, $\Sigma_m \dots \Sigma_1$ is a suffix of $\Sigma'_n \dots \Sigma'_1$ if and only if $m \leq n$ and $\Sigma_i \supseteq \Sigma'_i$ holds for every $i \in \{1, \dots, m\}$.

Equipped with these suffix orderings, which serve as abstraction relations, we can define the safety invariant. This is a predicate over a stack, written *safe s*. It is inductively defined as follows:

$$\mathit{safe}\ \varepsilon \qquad\qquad \frac{\begin{array}{c}\mathit{pastSymbols}(\sigma) \text{ is a suffix of } \mathit{symbols}(s)\\ \mathit{pastStates}(\sigma) \text{ is a suffix of } \mathit{states}(s)\\ \mathit{safe}\ s\end{array}}{\mathit{safe}\ s(\sigma, v)}$$

A stack $s(\sigma, v)$ is safe if (a) the annotations $\mathit{pastSymbols}(\sigma)$ and $\mathit{pastStates}(\sigma)$ associated with the current state σ are correct approximate descriptions of the tail s of the stack and (b) the tail s is itself safe. Less formally, $\mathit{pastSymbols}(\sigma)$ and $\mathit{pastStates}(\sigma)$ are static descriptions of a suffix of the stack (i.e., the part of the stack that is closest to the top). The rest of the stack, beyond this statically known suffix, is unknown. Nevertheless, this information is sufficient (if validation succeeds) to show that internal errors cannot occur: for instance, it guarantees that, whenever we attempt to pop k cells off the stack, at least k cells are present.

Now, in order to ensure that *safe s* is indeed an invariant of the interpreter, the validator must check that the annotations $\mathit{pastSymbols}(\sigma)$ and $\mathit{pastStates}(\sigma)$ are consistent. Furthermore, the validator must verify a few extra conditions which, together with the invariant, ensure safety. The safety validator checks that the following properties are satisfied:

1. For every transition, labeled X, of a state σ to a new state σ',
   - $\mathit{pastSymbols}(\sigma')$ is a suffix of $\mathit{pastSymbols}(\sigma)\mathit{incoming}(\sigma)$,
   - $\mathit{pastStates}(\sigma')$ is a suffix of $\mathit{pastStates}(\sigma)\{\sigma\}$.
2. For every state σ that has an action of the form $\mathit{reduce}\ A \longrightarrow \alpha\ \{f\}$,
   - α is a suffix of $\mathit{pastSymbols}(\sigma)\mathit{incoming}(\sigma)$,
   - If $\mathit{pastStates}(\sigma)\{\sigma\}$ is $\Sigma_n \dots \Sigma_0$ and if the length of α is k, then for every state $\sigma' \in \Sigma_k$, the goto table is defined at (σ', A). (If k is greater than n, take Σ_k to be the set of all states.)
3. For every state σ that has an *accept* action,
   - $\sigma \neq \mathit{init}$,
   - $\mathit{incoming}(\sigma) = S$,
   - $\mathit{pastStates}(\sigma) = \{\mathit{init}\}$.

Thanks to the finiteness of the alphabets, these conditions are clearly and efficiently decidable.

These conditions do not depend in any way on the manner in which lookahead is exploited to determine the next action. In other words, an $LR(1)$ automaton is safe if and only if the underlying non-deterministic $LR(0)$ automaton is safe. Thus, the safety validator is insensitive to which method was used to construct the $LR(1)$ automaton.

3.3 Completeness

The completeness theorem states that if the automaton passes the *completeness validator* (which we describe further on), if a prefix w of the input is valid, and if enough fuel is supplied, then the parser accepts w and constructs a correct semantic value. (The theorem also allows for the possibility that the parser might encounter an internal error. This concern was dealt with in §3.2, so we need not worry about it here.)

Theorem 3 (Completeness). *If the criteria enforced by the completeness validator are satisfied and if $S \xrightarrow{v} w$ holds, then there exists n_0 such that for all ω and for all $n \geq n_0$, either $parse(w\omega, n) = \lightning$ or $parse(w\omega, n) = parsed\ v\ \omega$.*

The Coq version of this result is in fact more precise. We prove that a suitable value of n_0 is the size of the derivation tree for the hypothesis $S \xrightarrow{v} w$, and we prove that this is the least suitable value, that is, $n < n_0$ implies $parse(w\omega, n) =$ *out-of-fuel*.

In order to guarantee that the automaton is complete, the validator must check that each state has "enough" permitted actions for every valid input to be eventually accepted. But how do we know, in a state σ, which actions should be permitted? We can answer this question if we know which set of $LR(1)$ items is associated with σ. Recall that an *item* is a quadruple $A \longrightarrow \alpha_1 \bullet \alpha_2\ [a]$, where $A \longrightarrow \alpha_1\alpha_2\ \{f\}$ is a production and a is a terminal symbol. The intuitive meaning of an item is: "we have recognized α_1; we now hope to recognize α_2 and find that it is followed with a; if this happens, then we will be able to reduce the production $A \longrightarrow \alpha_1\alpha_2\ \{f\}$".

Our items are relative to an *augmented* grammar, where a virtual production $S' \longrightarrow S$ has been added. This means that we can have items of the form $S' \longrightarrow \bullet S\ [a]$ (these appear in the initial state of the automaton) and items of the form $S' \longrightarrow S \bullet\ [a]$ (these appear in the final, accepting state).

The parser generator knows which set of items is associated with each state of the automaton. We require that this information be transmitted to the completeness validator. (One could instead reconstruct this information, and one would not even need to prove the correctness of the algorithm that reconstructs it; but it is not clear what one would gain by doing so.)

With this information, the validator carries out two kinds of checks. First, it checks that each state σ has "enough" actions: that is, the presence of certain items in $items(\sigma)$ implies that certain actions must be permitted. Second, it checks that the sets $items(\sigma)$ are *closed* and *consistent*: that is, the presence of certain items in $items(\sigma)$ implies that certain items must be present in $items(\sigma)$

(this is *closure*) and in *items*(σ'), where σ' ranges over the successor states of σ (this is *consistency*).

The definition of the closure property, which appears below, relies on the knowledge of the "*first*" sets, which in turn requires the knowledge of which non-terminal symbols are "*nullable*". It is well-known that "*first*" and "*nullable*" form the least fixed point of a certain system of positive equations. Again, we could have the validator compute this least fixed point; instead, we require that it be transmitted from the parser generator to the validator, and the validator just checks that it is a fixed point.

In summary, the annotations that the completeness validator requires (and that the parser generator must produce) are:

1. for each state σ, a set of items, written *items*(σ).
2. for each non-terminal symbol A, a set of terminal symbols *first*(A);
3. for each non-terminal symbol A, a Boolean value *nullable*(A).

The properties that the completeness validator enforces are:

1. "*first*" and "*nullable*" are fixed points of the standard defining equations.
2. For every state σ, the set *items*(σ) is closed, that is, the following implication holds:

$$\frac{\begin{array}{c} A \longrightarrow \alpha_1 \bullet A'\alpha_2\ [a] \in \mathit{items}(\sigma) \\ A' \longrightarrow \alpha'\ \{f'\} \text{ is a production} \\ a' \in \mathit{first}(\alpha_2 a) \end{array}}{A' \longrightarrow\ \bullet \alpha'\ [a'] \in \mathit{items}(\sigma)}$$

3. For every state σ, if $A \longrightarrow \alpha \bullet\ [a] \in \mathit{items}(\sigma)$, where $A \neq S'$, then the action table maps (σ, a) to *reduce* $A \longrightarrow \alpha\ \{f\}$.
4. For every state σ, if $A \longrightarrow \alpha_1 \bullet a\alpha_2\ [a'] \in \mathit{items}(\sigma)$, then the action table maps (σ, a) to *shift* σ', for some state σ' such that:

$$A \longrightarrow \alpha_1 a \bullet \alpha_2\ [a'] \in \mathit{items}(\sigma')$$

5. For every state σ, if $A \longrightarrow \alpha_1 \bullet A'\alpha_2\ [a'] \in \mathit{items}(\sigma)$, then the goto table either is undefined at (σ, A') or maps (σ, A') to some state σ' such that:

$$A \longrightarrow \alpha_1 A' \bullet \alpha_2\ [a'] \in \mathit{items}(\sigma')$$

6. For every terminal symbol a, we have $S' \longrightarrow\ \bullet S\ [a] \in \mathit{items}(\mathit{init})$.
7. For every state σ, if $S' \longrightarrow S \bullet\ [a] \in \mathit{items}(\sigma)$, then σ has a default *accept* action.

These conditions are clearly decidable. In order to achieve reasonable efficiency, we represent items in a compact way: first, we group items that have a common $LR(0)$ core; second, we use a natural number to indicate the position of the "bullet". Thus, in the end, we manipulate triples of a production identifier p, an integer index into the right-hand side of p, and a set of terminal symbols.

Furthermore, we use the standard library `FSets` [6], which implements finite sets in terms of balanced binary search trees, in order to represent sets of items.

The various known methods for constructing $LR(1)$ automata differ only in how they decide to merge, or not to merge, certain states that have a common $LR(0)$ core. The completeness validator is insensitive to this aspect: as long as no conflict arises due to excessive merging, an arbitrary merging strategy can be employed.

There is a relatively simple intuition behind the proof of Theorem 3. Suppose an oracle gives us a proof of $S \xrightarrow{v} w$, that is, a parse tree. Then, the parser, confronted with the input w, behaves in a very predictable way: it effectively performs a depth-first traversal of this parse tree. When the parser performs a *shift* action, it visits a (terminal) leaf of the parse tree; when it performs a reduce action, it visits a (non-terminal) node of the parse tree. At any time, the parser's stack encodes the path that leads from the root of the tree down to the current node of the traversal, and holds the semantic values associated with parse tree nodes that have been fully processed, but whose parent has not been fully processed yet. At any time, the unconsumed input corresponds to the fringe of the part of the parse tree that has not been traversed yet.

This invariant allows us to prove that the parser cannot reject the input $w\omega$. Instead, it keeps shifting and reducing until it has traversed the entire parse tree. At this point, it has consumed exactly w, and it must accept and produce exactly v. Note that there is no need to implement an "oracle". We simply prove that if *there exists* a parse tree, then the parser behaves *as if* it were traversing this parse tree, and accepts at the end.

In order to make this intuition precise, we define an invariant that relates the stack, the unconsumed input, and a path in the parse tree provided by the "oracle". The definition of this invariant is quite technical, but can be summed up as follows. There exists a path in the "oracle" parse tree such that:

1. The path begins at the root of the parse tree.
2. For each node in this path, the children of this node can be divided in three consecutive segments (or, at the last node in the path, in two segments):
   - children that have already been visited: the semantic value associated with each of these children is stored in a stack cell;
   - the child that is being visited (absent if we are at the bottom of the path): this child is the next node in the path;
   - children that will be visited in the future: their fringes correspond to segments of the unconsumed input stream.
3. The unconsumed input begins with the concatenation of the fringes of the "unvisited children" of all nodes in the path.
4. The sequence of all stack cells is in one-to-one correspondence with the concatenation of the sequences of "visited children" of all nodes in the path.
5. As per the previous item, each stack cell (σ, v) is associated with a certain child y of a certain node x in the path. Then, the semantic value carried by the node y must be precisely v. Furthermore, if the node x is labeled with the production $A \longrightarrow \alpha_1 X \alpha_2$ and if the child y corresponds to the symbol X,

then the item $A \longrightarrow \alpha_1 X \bullet \alpha_2$ $[a]$ appears in *items*(σ), where a is the first symbol in the partial fringe that begins "after" the node x.

During parsing, this path evolves in the following ways:

1. Between two actions: if the first unvisited child of the last node of the path exists and is not a terminal leaf, then the next action of the automaton will take place under this child. It is then necessary to extend the path: this child becomes the last node of the path. This extension process is repeated as many times as possible.
2. When shifting: if the first unvisited child of the last node exists and is a terminal leaf, then the next action must be a *shift* action. This child is considered visited and transferred to the first segment of children of the last node. The path itself is unchanged.
3. When reducing: if the last node of the path has no unvisited child, then the next action must be a *reduce* action for the production that corresponds to this node. Then, the path is shortened by one node: that is, if x and y are the last two nodes in the path, then x becomes the last node in the path, and y becomes the last visited child of x.

3.4 Unambiguity

It is easy to prove the following result.

Theorem 4. *Suppose there exists a token. If the criteria enforced by the safety and completeness validators are satisfied, then the grammar is unambiguous.*

The proof goes as follows. Suppose that the automaton is safe and complete. Suppose further that, for some word w, both $S \xrightarrow{v_1} w$ and $S \xrightarrow{v_2} w$ hold. By the safety hypothesis, the parser never encounters an internal error $\lightning$. Thus, by the completeness hypothesis, given a sufficiently large amount of fuel, the parser, applied to $w\omega$ (where ω is arbitrary), must produce v_1, and by a similar argument, must produce v_2. However, our automata are deterministic by definition: *parse* is a function. Thus, v_1 and v_2 must coincide.

4 Coq Formalization

All of the results presented in this paper were mechanized using the Coq 8.3pl1 proof assistant. The Coq formalization is fairly close to the definitions, theorems and proofs outlined in this paper.

We use of dependent types to support semantic values and semantic actions whose types are functions of the corresponding symbols and productions. This enables us to support user-supplied semantic actions with essentially no proof overhead compared with a "pure" parser that only produces a parse tree.

We make good use of Coq's module system: the validator and the interpreter are functors parameterized over abstract types for terminal and non-terminal symbols. The only constraints over these two types of symbols are that they must be finite (so that it is possible to decide universal quantifications over symbols) and they must come with a decidable total order (so that they can be used in conjunction with the `FSets` library).

The Coq formalization is pleasantly small: about 2500 lines, excluding comments. The executable specifications of the safety validator and the completeness validator are about 200 lines each. The proofs of soundness, safety, and completeness account for 200, 500, and 700 lines, respectively.

5 Experimentation on a C99 Parser

The grammar. Our starting point is the context-free grammar given in Annex A of the ISO C99 standard [9]. We omit the productions that support unprototyped "K&R-style" function definitions, since such old-style definitions are considered "an obsolescent feature" [9, section 6.11.7] and are not supported by subsequent passes of the CompCert compiler.

Both this slightly simplified C grammar and the original ISO C99 grammar are ambiguous. There are three distinct sources of ambiguity.

The first source of ambiguity is the classic "dangling-else" problem, which introduces a shift-reduce conflict in the $LR(1)$ automaton. We eliminated this conflict by a slight modification to the grammar, distinguishing two non-terminal symbols for statements: one that prohibits `if` statements without an `else` part, and the other that permits them.

The second source of ambiguity is the well-known problem with typedef names (identifiers bound to a type by a `typedef` declaration), which must be distinguished from other identifiers. For example, "`a * b;`" is to be parsed as a declaration of a variable `b` of type "pointer to `a`" if `a` is a typedef name, but stands for the multiplication of `a` by `b` otherwise. To avoid major ambiguities in the grammar, it is mandatory to use two distinct terminal symbols, *typedef-name* and *variable-name*, and rely on the lexer to classify identifiers into one of these two terminal symbols. The traditional way of doing so, affectionately referred to as "the lexer hack", is to have the semantic actions of the parser maintain a table of typedef names currently in scope, and to have the lexer consult this table to classify identifiers. We were reluctant to perform such side effects within the semantic actions of our verified parser. Instead, like Padioleau [16], we interpose a "pre-parser" between the lexer and the verified parser, whose sole purpose is to keep track of typedef names currently in scope and classify identifiers as either *typedef-name* or *variable-name* in the token stream that feeds the verified parser. For simplicity of implementation, the pre-parser is actually a full-fledged but unverified C99 parser that implements the standard "lexer hack" scheme.

The third and last source of ambiguity is also related to typedef names, but more subtle. Consider the declaration "`int f(int (a));`" where `a` is a typedef name. It can be read as "a function `f` with one parameter named `a` of type `int`", but also as "a function `f` with one anonymous parameter of function type `a` $\rightarrow$ `int`". The original ISO C99 standard leaves this ambiguity open, but Technical

Corrigendum 2 specifies that the second interpretation is the correct one [9, clause 6.7.5.3(11)]. Again, we rely on our pre-parser to correctly resolve this ambiguity (via well-chosen precedence annotations) and to ensure that identifiers in binding positions are correctly classified as typedef names (if previously bound by `typedef` and not to be redeclared, as in the example above) or as variable names (in all other cases, even if previously bound by `typedef`).

Generating the parser. We use the Menhir parser generator [19] modified to produce not only an $LR(1)$ automaton but also a representation of the source grammar as well as the various annotations needed by the validator. All outputs are produced as Coq terms and definitions that can be directly read into Coq. The modifications to Menhir are small (less than 500 lines of Caml code) and reside in just one new module.

Our modified C99 grammar comprises 87 terminal symbols, 72 non-terminal symbols, and 263 productions. The $LR(1)$ automaton generated by Menhir using Pager's method contains 505 states. (The grammar is in fact $LALR$, and the automaton produced by Menhir is indeed identical to the $LALR$ automaton that would be produced by an $LALR$ parser generator.) The generated Coq file is approximately 4.5 Mbytes long, of which 6% correspond to the automaton, 2% to the description of the grammar, and the remaining 92% are annotations (item sets, mostly).

Validating the parser. Running the validator on this output, using Coq's built-in virtual machine execution engine (the `Eval vm_compute` tactic), takes 19 seconds: 4s to validate safety and 15s to validate completeness[5]. Coq takes an additional 32s to read and type-check the file generated by Menhir, for a total processing time of 51s. While not negligible, these validation times are acceptable in practice. In order to further reduce the validation time, one could probably use improved data structures or extract the validator to natively-compiled OCaml code; but there is no pressing need.

Running the parser. With some elbow grease, we were able to replace CompCert 1.9's unverified parser (an $LALR$ automaton produced by OCamlYacc) with our new verified parser. The verified parser runs about 5 times slower than the old one, increasing overall compilation times by about 20%. There are two major reasons for this slowdown. One is that we effectively parse the input twice: once in the pre-parser, to track typedef names, and once "for good" in the verified parser. Another reason is that the interpreter that executes the verified parser is written in Coq, then extracted to Caml, and performs redundant runtime checks compared with OCamlYacc's execution engine, which is coded in C and performs no runtime checks whatsoever. (We discuss the issue of redundant checks in §7.)

[5] All timings were measured on a Core i7 3.4GHz processor, using a single core.

6 Related Work

Although parsing is a classic and extremely well-studied topic, the construction of verified parsers seems to have received relatively little interest.

Pottier and Régis-Gianas [20] show that, for a fixed $LR(1)$ automaton, the inductive invariant that describes the stack and guarantees safety (§3.2) can be expressed as a generalized algebraic data type (GADT). They show that if one constructs a parser by *specializing* the interpreter for this automaton, then a type-checker equipped with GADTs can verify the safety of this parser. In addition to a safety guarantee, this approach yields a performance gain with respect to an ML implementation, because the weaker type system of ML imposes the use of redundant tags and dynamic checks (e.g. stack cells must be redundantly tagged with *nil* or *cons*). Here, the interpreter is *generic*, and, even though it is provably safe, it does perform redundant dynamic checks. (We discuss this issue in §7.)

Barthwal and Norrish [4] and Barthwal [3] use the HOL4 proof assistant to formalize SLR [5] parsing. For a context-free grammar, they construct an SLR parser, and are able to prove it sound and complete: the guarantees that they obtain are analogous to our Theorems 1 and 3. Like us, they omit a proof that the parser terminates when presented with an illegal input. (We discuss this issue in §7.) Although their parsers are executable, they are probably not efficient, because the parser construction and parser execution phases are not distinguished in the usual manner: in particular, while the parser is running, states are still represented as sets of $LR(1)$ items. Because Barthwal and Norrish formalize the parser construction process, their formalization is relatively heavyweight and represents over 20 000 lines of definitions and proofs. In contrast, because we rely on a validator, our approach is more lightweight (2500 lines). It is also more versatile: we can validate $LR(1)$ automata constructed by any means, including $LR(0)$, SLR, $LALR$, Pager's method, and Knuth's canonical $LR(1)$ construction. We believe that this versatility is important in practice: for instance, we have verified that the C99 grammar is $LALR$ but not SLR. One disadvantage of our approach is that we cannot exclude the possibility that the parser generator (which is not verified) fails or produces an incorrect automaton. Fortunately, this problem is detected at validation time. In our application to CompCert, it is detected when CompCert itself is built, that is, before CompCert is distributed to its users.

Parsing Expression Grammars (PEGs) are a declarative formalism for specifying recursive descent parsers. Ford [7,8] and other authors [24] have investigated their use as an alternative to the more traditional and better established context-free grammars. Koprowski and Binsztok [13] formalize the semantics of PEGs, extended with semantic actions, in Coq. They implement a well-formedness check, which ensures that the grammar is not left-recursive. Under this assumption, they are able to prove that a straightforward (non-memoizing) PEG interpreter is terminating. The soundness and completeness of the interpreter are immediate, because the interpreter is just a functional version of the semantics of PEGs, which is originally presented under a relational form. Wisnesky *et al.* [25] implement a verified packrat parser (that is, a PEG parser that achieves linear time and space complexity via memoization) using the experimental programming language Ynot, which is itself embedded within Coq. Because Ynot is a Hoare logic for partial correctness, the parser is not proved to terminate.

We are definitely not the first to embrace *a posteriori* validation as an effective way to obtain correctness guarantees for compilers and program generators. This idea goes back at least to Samet's 1975 Ph.D. thesis [21] and was further developed under the name *translation validation* by Pnueli *et al.* [18] and by Necula [15]. Tristan and Leroy exploit the idea that formally-verified validators for advanced compiler optimizations provide soundness guarantees as strong as direct compiler verification [23]. Another example of a formally-verified validator is the JVM bytecode verifier of Klein and Nipkow [11].

7 Conclusions and Future Work

The approach to high-assurance parsing that we have developed, based on *a posteriori* validation of an untrusted parser generator, appears effective so far. The validation algorithms are simple enough that they could be integrated into production parser generators and used as sanity checkers and debugging aids, even in contexts where strong assurance is not required.

This work can be extended in several directions, including proving additional properties of our $LR(1)$ parser, improving its efficiency, and extending our work to more expressive parsing formalisms. We now review some of these directions.

We have not proved that our parser terminates. Completeness (Theorem 3) implies that it terminates when supplied with a valid input. However, there remains a possibility that it might diverge when faced with an invalid input. In fact, we have proof that the properties enforced by the safety and completeness validators are not sufficient to ensure termination. So, more requirements must be enforced, but we are not sure, at present, what these requirements should be. Aho and Ullman prove that *canonical* $LR(1)$ parsers terminate [1, Theorem 5.13]. Their argument exploits the following property of canonical $LR(1)$ parsers: "as soon as [the consumed input and] the first input symbol of the remaining input are such that no possible suffix could yield a sentence in $\mathcal{L}(\mathcal{G})$, the parser will report error". This property, however, is *not* true of non-canonical $LR(1)$ parsers. By merging several states of the canonical automaton that have a common $LR(0)$ core, the non-canonical construction methods introduce spurious reductions: a non-canonical automaton can perform a few extra reductions before it detects an error. Thus, Aho and Ullman's proof does not seem to apply to non-canonical $LR(1)$ parsers.

We have not proved that our parser does not peek one token past the end of the desired input. We claim that this property holds: the automata produced by Menhir use default actions for this purpose (see Remark 1 in §2.4). However, at present, we cannot even *state* this property, because it is an intensional property of the function *parse*: "if $\mathit{parse}(\omega, n) = \mathit{parsed}\ v\ \omega'$, then ω' has not been forced".

In order to allow this property to be stated, one approach would be to reformulate the parser so that it no longer has access to the token stream, but must instead interact explicitly with the producer of the token stream, by producing "peek" or "discard" requests together with a continuation.

We have defined a "cautious" interpreter, which can in principle encounter an internal error, and we have proved, after the fact, that (if the safety validator is satisfied, then) this situation never arises. This allows us to separate the definition of the interpreter and the safety argument. A potentially significant drawback of this approach is that it entails a performance penalty at runtime: even though we have a proof that the dynamic checks performed by the cautious interpreter are redundant, these checks are still present in the code, and slow down the parser. An anonymous reviewer pointed out that, without modifying any of our existing code, it should be possible to define an "optimistic" interpreter, which performs no runtime checks, and is subject to the precondition that the cautious interpreter does not fail. Sozeau's "`Program`" extensions [22] could be used to facilitate the construction of this optimistic interpreter. In the end, only the optimistic interpreter would be executed, and the cautious interpreter would serve only as part of the proof. We would like to thank this reviewer for this attractive idea, which we have not investigated yet.

Some of the hints that we require are redundant. In particular, *pastSymbols* (§3.2) is entirely redundant, since it can in fact be deduced from *pastStates* and *incoming*. This redundancy is due to historical reasons, and could be eliminated. This might help speed up the type-checking and validation of the annotations.

Beyond $LR(1)$ parsing, we believe that validation techniques can apply to other parsing formalisms. It would be particularly interesting to study GLR, for which we hope that our validators could be re-used with minimal changes.

Our experience with the C99 grammar agrees with common wisdom on the importance of supporting precedence and associativity declarations in order to keep parser specifications concise and readable. It is well-known how to take these declarations into account when generating $LR(1)$ automata, but the resulting automata are no longer complete with respect to the grammar. How could we modify our definition of the meaning of a grammar so as to take precedence declarations into account? How could we then extend our validation algorithms?

References

1. Aho, A.V., Ullman, J.D.: The theory of parsing, translation, and compiling. Prentice Hall (1972)
2. Anderson, T., Eve, J., Horning, J.J.: Efficient $LR(1)$ parsers. Acta Informatica 2, 12–39 (1973)
3. Barthwal, A.: A formalisation of the theory of context-free languages in higher order logic. Ph.D. thesis, Australian National University (Dec 2010)
4. Barthwal, A., Norrish, M.: Verified, executable parsing. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 5502, pp. 160–174. Springer (2009)

5. DeRemer, F.L.: Simple $LR(k)$ grammars. Communications of the ACM 14(7), 453–460 (1971)
6. Filliâtre, J.C., Letouzey, P.: Functors for proofs and programs. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 2986, pp. 370–384. Springer (Mar 2004)
7. Ford, B.: Packrat parsing: simple, powerful, lazy, linear time. In: ACM International Conference on Functional Programming (ICFP). pp. 36–47 (Oct 2002)
8. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: ACM Symposium on Principles of Programming Languages (POPL). pp. 111–122 (Jan 2004)
9. ISO/IEC: Programming languages — C (2007), international standard ISO/IEC 9899:TC3
10. Jourdan, J.H., Pottier, F., Leroy, X.: Coq code for validating $LR(1)$ parsers, `http://www.eleves.ens.fr/home/jjourdan/parserValidator.tgz`
11. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Transactions on Programming Languages and Systems 28(4), 619–695 (2006)
12. Knuth, D.E.: On the translation of languages from left to right. Information & Control 8(6), 607–639 (Dec 1965)
13. Koprowski, A., Binsztok, H.: TRX: A formally verified parser interpreter. Logical Methods in Computer Science 7(2) (2011)
14. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
15. Necula, G.C.: Translation validation for an optimizing compiler. In: ACM Conference on Programming Language Design and Implementation (PLDI). pp. 83–95. ACM Press (2000)
16. Padioleau, Y.: Parsing C/C++ code without pre-processing. In: Compiler Construction. Lecture Notes in Computer Science, vol. 5501, pp. 109–125. Springer (2009)
17. Pager, D.: A practical general method for constructing $LR(k)$ parsers. Acta Informatica 7, 249–268 (1977)
18. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorithms for Construction and Analysis of Systems, TACAS '98. Lecture Notes in Computer Science, vol. 1384, pp. 151–166. Springer (1998)
19. Pottier, F., Régis-Gianas, Y.: The Menhir parser generator, `http://gallium.inria.fr/~fpottier/menhir/`
20. Pottier, F., Régis-Gianas, Y.: Towards efficient, typed LR parsers. Electronic Notes in Theoretical Computer Science 148(2), 155–180 (2006)
21. Samet, H.: Automatically Proving the Correctness of Translations Involving Optimized Code. Ph.D. thesis, Stanford University (1975)
22. Sozeau, M.: Program-ing finger trees in Coq. In: ACM International Conference on Functional Programming (ICFP). pp. 13–24 (Sep 2007)
23. Tristan, J.B., Leroy, X.: A simple, verified validator for software pipelining. In: ACM Symposium on Principles of Programming Languages (POPL). pp. 83–92. ACM Press (2010)
24. Warth, A., Douglass, J.R., Millstein, T.D.: Packrat parsers can support left recursion. In: ACM Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM). pp. 103–110 (Jan 2008)
25. Wisnesky, R., Malecha, G., Morrisett, G.: Certified web services in Ynot. In: Workshop on Automated Specification and Verification of Web Systems (Jul 2009)