Verified, Executable Parsing

Aditi Barthwal[1] and Michael Norrish[2]

[1] Australian National University
Aditi.Barthwal@anu.edu.au
[2] Canberra Research Lab., NICTA
Michael.Norrish@nicta.com.au

Abstract. We describe the mechanisation of an SLR parser produced by a parser generator, covering background properties of context-free languages and grammars, as well as the construction of an SLR automaton. Among the various properties proved about the parser we show, in particular, *soundness*: if the parser results in a parse tree on a given input, then the parse tree is valid with respect to the grammar, and the leaves of the parse tree match the input; *completeness*: if the input is in the language of the grammar then the parser constructs the correct parse tree for the input with respect to the grammar; and *non-ambiguity*: grammars successfully converted to SLR automata are unambiguous.

We also develop versions of the algorithms that are executable by automatic translation from HOL to SML. These alternative versions of the algorithms require some interesting termination proofs.

1 Introduction

The (context-free) parsing problem is one of determining whether or not a string of terminal symbols belongs to a language that has been specified by means of a context-free grammar. In addition, we imagine that the input is to be processed by some later form of analysis, *e.g.*, a compiler. Therefore, we wish to generate the parse tree that demonstrates this membership when the string is in the language, rather than just a yes/no verdict.

The parsing problem can be solved in a general way for large classes of grammars through the construction of deterministic push-down automata. Given any grammar in the acceptable class, the application of one function produces an automaton embodying the grammar. This automaton then analyses its input, producing an appropriate parse tree. The particular function we have chosen to formally characterise and verify produces what is known as an SLR automaton.

Thus, at a high level, our task is to specify and verify two functions

```
slrmac : grammar -> automaton option
parse : automaton -> token list -> ptree option
```

The `slrmac` function returns `SOME m` if the grammar is in the SLR class, and `NONE` otherwise. The `parse` function uses the machine `m` to consume the input and produce a parse tree for the input string, returning `NONE` in case of a failure. A parser generator

G. Castagna (Ed.): ESOP 2009, LNCS 5502, pp. 160–174, 2009.

is used to produce such a `parse` function for any context-free grammar. In this paper we will concentrate on implementation and verification of the `parse` function.

In the rest of the paper, we will describe the types and functions that appear above. In Section 1.1, we describe grammars and their properties. In Section 1.2, we describe the type of SLR automata, and the type of the results. In Section 1.3, we describe the construction of automata from input grammars. We are then in a position to verify important properties about these functions. Our theorems are described in Section 2. Finally, we also wish to be able to turn our verified HOL functions into functions that can be executed in SML. To do this, a number of definitions that have rather abstract or "semantic" characterisations need to be shown to have executable equivalents. The derivation of executable forms is described in Section 3.

Literature and Technology. Being one of our field's earliest examples of theory leading to successful practice, parsing and language theory has a large literature. On the other hand, we are not aware of any existing work on a mechanised theory of parsing. Our mechanisation has been performed in the HOL4 system [2,5], and has been inspired principally by Hopcroft and Ullman's standard text [3].

Parsers as External Proof Oracles. If an external, potentially untrusted, tool were to generate the parse tree for a given string, it would be easy to verify that this parse tree was indeed valid for the given grammar. The parse tree would be serving as a proof that the input string was indeed in the grammar's language, and the trusted infrastructure need only check that proof. It is natural then to ask what additional value a verified parser-generator might provide. Apart from the intellectual appeal in mechanising interesting mathematics, we believe there is at least one pragmatic benefit: if the (verified) construction of an SLR automaton succeeds, one has a proof that the grammar in question is unambiguous. When a parse is produced by the automaton, one knows that no other parse is possible.

1.1 Context-Free Grammars

A context-free grammar (CFG) is represented in HOL using the following type definitions:

```
symbol = TS of string | NTS of string

rule = rule of string => symbol list

grammar = G of rule list => string
```

(The => arrow indicates curried arguments to an algebraic type's constructor. Thus, the `rule` constructor is a term of type `string -> symbol list -> rule`. We use lists rather than sets for the grammar's rules for ease of later translation to SML, and to avoid frequent finite-ness side conditions.)

A rule is a mapping from a string to a symbol list, where the string is interpreted as a non-terminal. Similarly, a grammar consists of a list of rules and a string giving the start

symbol. Traditional presentations of grammars often include separate sets corresponding to the grammar's terminals and non-terminals. We extract these sets with functions `terminals` and `nonTerminals` respectively.

Definition 1. *A list of symbols (or* sentential form*)* s derives t *in a single step if* s *is of the form* $\alpha A \gamma$*,* t *is of the form* $\alpha\beta\gamma$*, and if* $A \rightarrow \beta$ *is one of the rules in the grammar. In HOL:*

```
derives g sf1 sf2 =
  ∃s1 s2 rhs N.
    (sf1 = s1 ++ [NTS N] ++ s2) ∧
    (sf2 = s1 ++ rhs ++ s2) ∧
    MEM (rule N rhs) (rules g)
```

(The infix ++ denotes list concatenation. The `MEM` *relation denotes list membership.)*

We can form the reflexive and transitive closure of a binary relation like `derives g` with the `^*` operator, written as a suffix. Thus, `(derives g)^* sf1 sf2` indicates that `sf2` is derived from `sf1` in zero or more steps, also denoted as `sf1` $\Rightarrow^*$ `sf2` w.r.t a grammar.

Later we will also use the rightmost derivation relation, `rderives`, and its closure.

Definition 2. *The* language *of a grammar consists of all the words that can be derived from the start symbol.*

```
language g =
  { tsl | (derives g)^* [NTS (startSym g)] tsl ∧
          EVERY isTmnlSym tsl }
```

(Predicate `isTmnlSym` *is true of a symbol if it is of the form* `TS s` *for some string* `s`*.* `EVERY` *checks that every element of a list satisfies the given predicate.)*

We also define the concept of nullability and relations for finding first sets and follow sets for a symbol as stated below. These notions are central when the actions for the SLR automaton are calculated (see Section 1.2).

Definition 3. *A list of symbols* α *is* nullable *iff* $\alpha \Rightarrow^* \epsilon$*:*

```
nullable g sl = (derives g)^* sl []
```

Definition 4. *The* first *set of a symbol is the set of terminals that can appear first in the sentential forms derivable from it:*

```
firstSet g sym =
  { (TS fst) | ∃rst.(derives g)^* [sym] (TS fst::rst) }
```

(:: *represents the list 'cons' operator.)*

Definition 5. *The* follow *set of a symbol* N *is the set of terminals that can occur after* N *in a sentential form derivable from any of the right-hand sides belonging to a rule in the grammar.*

```
followSet g N =
  { TS ts | ∃M rhs p s.
              MEM (rule M rhs) (rules g) ∧
              (derives g)^* rhs (p ++ [N;TS ts] ++ s) }
```

(This definition might be simplified by only considering derivations from the start symbol of the grammar. However, we choose to present it in the above way so it is compatible with our executable definition, which ignores reachability of non-terminals.)

Executable versions of these functions (which do not need to scan all possible derivations) are described in Section 3.1.

1.2 SLR Automata

An SLR machine is a push-down automaton where each state in the automaton corresponds to a list of *items*. An item $N \rightarrow \alpha \cdot \beta$, is a grammar rule that has been split in two by the dot ($\cdot$) marking the progress that has been made in recognising the given right-hand side ($\alpha\beta$). In HOL:

```
item = item of string => symbol list # symbol list
state = item list
```

In the mechanisation, an automaton state is a list of items, and the empty list represents an error state. The state of an execution is the current input, coupled with a stack of pairs of automaton states and parse trees. The root of each parse tree corresponds to a terminal symbol that has been shifted from the input, or to a non-terminal that has been produced through a reduction step.

Based on the next symbol in the input (we are implementing SLR with one symbol lookahead), and the state the parser is in, the parser will perform one of the following actions:

- `REDUCE`: the parser recognizes a valid handle on the stack and reduces it to the left-hand side of the rule
- `GOTO`: the parser shifts an input symbol on to the stack and goes to the indicated state
- `NA`: the parser throws an error

In our framework, the automaton is presented by two functions, `goto` and `reduce`. The `goto` function takes a `symbol` and a `state` as arguments and returns a new `state`. We have thus merged two tables in the traditional presentation: the shift table encoding information for terminals, and the goto table for non-terminals.

The `reduce` function takes a `symbol` and a `state` and returns a list of possible rules that can be reduced in the given state. When the machine has been constructed from an SLR grammar the list will always be empty or just one element long. If a reduction is to be performed for rule $N \rightarrow \alpha$, the symbols α are popped off the stack, revealing a state s_0. The non-terminal N is pushed onto the stack, and the machine shifts to the state given by `goto` applied to N and s_0.

Given a state and input symbol, the next action is a shift if the `goto` function returns a non-error state. The next action is a reduction if the `reduce` function returns a list containing one rule. The SLR construction ensures that both conditions can't be true simultaneously. If neither is true, the machine throws an error.

These functions are combined using a while combinator of type

```
('a -> bool) -> ('a -> 'a option) -> 'a ->
'a option option
```

The type `'a` is the type of the execution state. The first argument is a boolean condition on states specifying when the loop should continue. The second argument encodes the loop body, allowing for the possibility that the loop execution terminates abnormally (*e.g.* the parser detects a string not in the grammar's language). The third argument is the initial state. The result encodes normal termination, abnormal termination (`SOME NONE`) and failure to terminate (`NONE`).

1.3 Constructing the Parser

The architecture of the parser-construction process is shown in Figure 1. The first step in creating the SLR machine is to augment the grammar. The augmentation adds an extra rule that introduces a new start symbol and a marker (a terminal symbol) that appears at the end of all the words in the language of the grammar. The parser uses this rule for reduction exactly when it has accepted the input word. This ensures that the parser always 'spots' the end of input. The augmentor `auggr` is a function of type

```
grammar -> string -> string -> grammar option
```

We use `SOME g'` to return the augmented grammar `g'` when the symbols being introduced are 'fresh' (not part of the old grammar). Otherwise failure is indicated by returning `NONE`.

The `slrmac` function creates the `goto` and `reduce` functions which represent the three transition tables of the traditional presentation of an LR automaton. It checks that

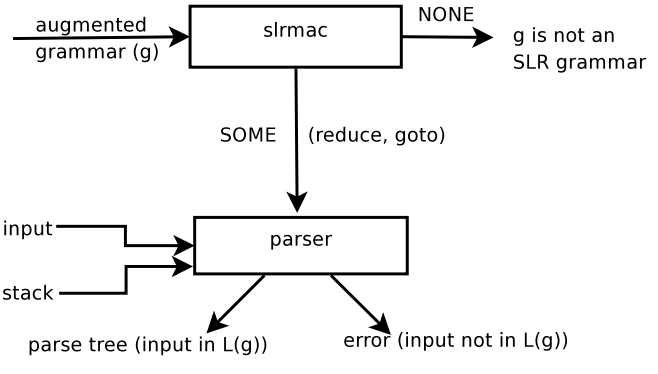

Fig. 1. Architecture of the Parser Construction Process

the functions don't produce any shift-reduce or reduce-reduce conflicts. If the functions pass this test, they can be passed onto the `parser` function which implements the machine (as described above in Section 1.2).

Building the Parsing Tables. The construction of the `goto` function is conceptually simple: let the result of applying `goto` to a state σ and the symbol s (terminal or non-terminal) be the list of items $N \rightarrow \alpha s \cdot \beta$, where $N \rightarrow \alpha \cdot s\beta$ is an element of σ. This behaviour is captured in the HOL function `moveDot`. Unfortunately, it is not sufficient.

When an item's dot is before a non-terminal, say $A \rightarrow \alpha \cdot B\beta$, this indicates that the parser expects to parse the non-terminal (B) next. To ensure the item set contains all possible rules the parser may be in the midst of parsing, it must additionally include all items describing how B itself will be parsed. If there are rules for B that themselves have non-terminals as the first element of a RHS, then those non-terminals' items must also be included. Thus we must take a closure: repeatedly including all referenced non-terminals until we reach a fix-point.

The final `goto` function is calculated by `nextState` (which gets access to the input grammar). The new state is computed by moving the dot over all the items in the current state that have the input symbol after the dot, and then taking the closure.

```
nextState g itl sym = closure g (moveDot itl sym)
```

The other table we must compute is `reduce`. This really is simple: for every complete item (of the form $N \rightarrow \alpha\cdot$) in a state, return the rule $N \rightarrow \alpha$ if the input symbol is in the follow set of N. Because we use the entire follow set of N, we are computing an SLR machine. If we didn't use a follow set at all, and always reduced on complete items, we would be implementing an LR(0) parser. If we computed follow sets for states that depended on where a non-terminal had been used, we would be implementing an LALR parser.

Checking for Conflicts. When `slrmac` has constructed the functions `goto` and `reduce`, it then checks them for possible shift-reduce or reduce-reduce conflicts. Checking for such an error in a given state on a given symbol is done by the `noError` function:

```
noError (go,rd) sym st =
  case rd st sym of
     []  -> T
  || [r] -> (go st sym = [])
  || otherwise -> F
```

The `slrmac` function then tests `noError` on all reachable states in the automaton, and for all possible terminal symbols. This is easy to express logically:

```
okSlr g initState =
  ∀syms state tok.
    trans g (initState, syms) = SOME state ⟹
    noError (goto g, reduce g) tok state
```

where `trans g` iterates `goto g` over a sequence of symbols to find the resulting state (if any). Hopcroft and Ullman call this function δ.

Expressing this check executably is discussed in Section 3.

Putting it all Together. The `parser` function is as given in Figure 1.

```
parser (initState, eof, oldS) m sl  =
   let out = mwhile (¬ ∘ exitCond eof oldS)
                    (λs.parse m s) (init initState sl)
   in
     case out of
          NONE -> NONE
        || SOME (SOME (sl',[(state,ptree)],csl')) ->
             SOME (SOME ptree)
        || SOME NONE -> SOME NONE
        || SOME _ -> SOME NONE
```

The `parse` function implements a single step of the SLR machine (Section 1.2). `init` provides the initial execution state to get this process started. The `exitCond` function is true of an execution state if the stack consists of just the non-augmented grammar's start symbol, and if the input consists of just the `eof` token. The while combinator `mwhile` (Section 1.2) repeatedly performs the `parse` step until `exitCond` is true.

2 Proofs

We now have a parser generator formally specified in HOL. To verify that our specification is indeed correct, we would like to demonstrate that the language accepted by the automaton is the same as the language defined by the grammar. This goal is naturally split into two inclusion results: that everything accepted by the machine is in the language ("soundness"), and that everything in the language is accepted by the machine ("completeness").

Before we delve into the proofs, we describe what it means to be a valid parse tree with respect to a grammar:

```
(validptree g (Node n ptl) =
  MEM (rule n (getSymbols ptl)) (rules g) ∧
  (∀e. MEM e ptl ∧ isNode e ⟹ validptree g e)) ∧
(validptree g (Leaf tm) = F)
```

Here, `getSymbols` gives the list of symbols at the roots of a list of trees. Thus, a tree is valid with respect to a grammar if there is a rule in the grammar that corresponds to the root node deriving the roots of its sub-trees, and if (recursively) all the sub-trees are also valid.

The proofs to come also depend on a number of simple invariants on the state of a parse execution:

– **parser_inv** states implementation-specific properties about the stack. These properties ensure the items in each of the state on the stack correspond to some grammar rule (validStates) and that the initial start state is never popped off from the stack.

```
parser_inv g csl = validStates g csl ∧ ¬NULL csl
```

– The SLR automaton works by computing valid items for each viable prefix. Predicate **validItem_inv** asserts that each of the states contains only those items that are valid for the viable prefix γ, which is the string of symbols that has been pushed on to the stack to reach that state (stk).

```
validItem_inv g initState stk =
  ∀stk'.
    IS_PREFIX stk stk' ∧ ¬NULL stk'
   ⟹
    trans g (initState, stackSyms stk') =
      SOME (topState stk')
```

2.1 Validity of the Parse Tree Generated

If the parser results in a parse tree, the tree is valid with respect to the grammar for which the parser was generated. Alternatively, the parse tree was built using rules present in the given grammar.

Below we abbreviate validptree_inv for conditions which state that for all the non-terminals on the stack, the associated parse trees are valid with respect to the given grammar. We prove that this property is preserved by the parse function, which takes a single step of the execution. By induction over the while-loop, if the parser is able to reduce the stack symbols to the start symbol, then the corresponding parse tree must be valid as well.

Theorem 1

```
∀g sl stl.
   auggr g s eof = SOME ag ∧ slrmac ag = SOME m ∧
   parser_inv ag csl ∧ validptree_inv g stl ∧
   parser (initState, eof, oldS) (SOME m) sl =
     SOME (SOME tree)
⟹
   validptree ag tree
```

2.2 Equivalence of the Output Parse Tree and the Input String Parsed

The main predicate of interest here is the leaves_eq_inv. Below it abbreviates conditions which assert that at each state the leaves of the tree are equal to the parsed string. This ensures that the grammar rules being applied to form the parse tree, correspond to the input string being parsed and the leaves of the resulting parse tree are equal to the original input string.

Theorem 2

```
∀m g s eof sl csl.
   auggr g s eof = SOME ag ∧ slrmac ag = SOME m ∧
   parser_inv ag csl ∧ leaves_eq_inv sl sl [] ∧
   parser (initState, eof, startSym g) (SOME m) sl =
      SOME (SOME tree))
⟹
   (sl=leaves tree)
```

2.3 Soundness of the Parser

To prove soundness, we have to show that the input string for which a valid parse tree can be constructed, is in the language of the grammar.

Theorem 3

```
∀m g s eof sl csl.
  auggr g s eof = SOME ag ∧ slrmac ag = SOME m ∧
  parser_inv ag (stl, csl) ∧
  validptree_inv ag (stl, csl) ∧
  leaves_eq_inv sl sl [] ∧
  parser (initState, eof, startSym g) (SOME m) sl =
      SOME (SOME tree))
⟹
  sl ∈ language ag
```

In turn, this result depends on a simple result stating the equivalence of being able to derive a sentential form and having a valid parse tree with that form as its leaves.

2.4 Completeness of the Parser

To show completeness, we have to prove that if a string is in the language of a grammar then the parser will terminate with a parse tree. Soundness (Theorem 3) already ensures the validity of the output tree. We assume that the grammar does not have useless non-terminals, *i.e.* all the non-terminal symbols generate some terminal string ('generates a word', `gaw`). We earlier proved that removing useless symbols does not affect the language of a grammar, so we might extend `slrmac` to do this for us, or just have it report an error if given a grammar containing useless non-terminals.

Theorem 4

```
   auggr g st eof = SOME ag ∧ sl ∈ language ag ∧
   slrmac ag = SOME m ∧
   (∀nt. nt ∈ nonTerminals ag ⟹ gaw ag nt)
⟹
   ∃tree.
   parser (initState, eof, startSym g) (SOME m) sl =
      SOME (SOME tree)
```

This result has by far the most complicated proof in the mechanisation, and took a considerable proportion of the total time spent. Much of the time was spent casting about for a detailed version of the argument for LR(0) grammars in Hopcroft and Ullman [3, §10.7]. That argument specifies the construction of the automaton and continues:

> We claim that when M starts with w in $L(G)$ on its input and only s_0 on the stack, it will construct a rightmost derivation for w in reverse order. The only point still requiring proof...

Our eventual proof recasts this somewhat. We already have an (arbitrary) rightmost derivation for w by virtue of the fact that it is in $L(G)$. (We proved the lemma stating that any derivation of a word has a rightmost equivalent.) We then argue that the machine will take a sequence of steps that mirror this derivation.

We make the actual derivation concrete (it is a list of sentential forms), and write $R \vdash d \lhd sf_0 \rightarrow sf_1$ if d is a derivation of sf_1, starting at sf_0, and respecting derivation relation R (*i.e.*, R holds between each successive pair of elements in the list d).

Each sentential form is derived from its predecessor by the expansion of a nonterminal. When moving backwards through the derivation, this corresponds to a reduction step.

The crucial lemma supporting our proof states that if we have $\texttt{rderives}\ g \vdash d \lhd sf_0 \rightarrow w$, then there is a sequence of n `parse`-steps bringing the SLR automaton to a state where it is just about to perform the first reduction of the derivation d. This is by induction on d. This result in turn relies on knowing that when the current *handle*, or RHS of the next reduction, is still partly or completely in the input, the machine will perform a sequence of shift moves in order to bring the handle onto the stack.

All of these results depend on the invariants already described, and the fact that the automaton is SLR. For example, in the last lemma: if we know that a shift is possible, then we also know that a reduction is not.

2.5 SLR Grammars Are Unambiguous

A grammar is unambiguous if for each string $w \in$ L(G), w has a unique rightmost derivation.

Definition 6. *A word w in the language of grammar g is represented by a derivation list starting from the start symbol of g and ending in w. A derivation for w is unique iff all possible derivation lists are identical.*

```
isUnambiguous g =
  ∀sl dl dl'.
    sl ∈ language g ∧
    rderives g ⊢ dl ◁ [NTS (startSym g)] → sl ∧
    rderives g ⊢ dl' ◁ [NTS (startSym g)] → sl ∧
   ⟹
    dl=dl'
```

Theorem 5

```
auggr g st eof = SOME ag ∧ slr ag = SOME m
     ⟹
isUnambiguous ag
```

A corollary of completeness and the fact that the SLR machine is deterministic.

3 An Executable Parser

For the most part, the HOL definitions turn out to be executable. However, for the sake of simplicity and clarity, many of our definitions were written in a style that favoured mathematical ease of expression. The use of existential quantifiers, and the reflexive and transitive closure in such definitions make them non-executable. Here we describe how the defined functions can be re-expressed in a way that makes them acceptable to HOL4's `emitML` technology. Our general approach was to take an existing function f, and define a new f`ML` constant. After proving termination for the typically complicated recursion equations defining f`ML`, we then had to show that f`ML`'s behaviour was equivalent to f's.

Would it save work to just use executable functions from the outset? Sadly no; the important thing about these executable functions is that they should compute some mathematical property. Proving that this is the case is the same problem as showing the equivalences we describe here.

In this section we describe our executable implementations of the non-executable, or "mathematical" HOL definitions. Even though the HOL versions were more tractable for proving properties such as our language inclusion results, there have been places where it was decided to value executability over succinctness of presentation.

3.1 Executable Calculation of Nullable Non-terminals

The executable counterpart of the `nullable` function is given below.

```
nullableML g sn [] = T ∧
nullableML g sn (TS ts::rest) = F ∧
nullableML g sn (NTS A::rest) =
  if (MEM (NTS A) sn) then F
  else
    EXISTS (nullableML g (NTS A::sn))
           (getRhs A (rules g)) ∧
    nullableML g sn rest
```

The `nullableML` function determines whether or not a list of symbols (a sentential form) can derive the empty string. When the string includes a terminal symbol, the result is false. When a non-terminal is encountered, we recursively determine if any of that non-terminal's RHSes might derive the empty string.

In order to ensure that this recursion terminates, we introduce a "seen" list and update this with the non-terminal that is being visited when we expand it. To then convince HOL that this function terminates, we must find a wellfounded relation on the

arguments of `nullableML`. Because a singleton list containing a non-terminal may expand into a list of symbols of arbitrary length, we cannot simply use the length of the sentential form as a measure. Instead we use the lexicographic combination:

```
measure (λ(g,sn). |nonTerminals g \ set sn|)
   LEX
measure LENGTH
```

We assert that either the number of symbols except the ones in the seen list decreases, or that the length of the sentential form decreases. The former corresponds to the first conjunct in the third clause in the definition while the latter takes care of the second conjunct.

The next step is to show the equivalence between the new HOL constants and the originals. Proving the equivalence requires showing the following two implications.

```
∀g sn sf. nullableML g sn sf ⟹ nullable g sf
```

```
∀g sf. nullable g sf ⟹ sn = [] ⟹ nullableML g sn sf
```

As previously outlined, for a sentential form to be nullable, it cannot have a terminal symbol. We look at the non-trivial case, *i.e.*when the sentential form itself is not empty. A sentential form $N_1 N_2 ... N_n$ is nullable iff the individual derivations for the Ns itself are nullable.

$$
\begin{aligned}
&\mathrm{N}_1 \Rightarrow^* \epsilon \\
&\mathrm{N}_2 \Rightarrow^* \epsilon \\
&\cdot \\
&\cdot \\
&\cdot \\
&\mathrm{N}_n \Rightarrow^* \epsilon
\end{aligned}
$$

`nullable` asserts the existence of *some* derivation from sf to ϵ. On the other hand, `nullableML` looks at a concrete derivation with a specific property, *i.e.*in each individual derivation, the symbols cannot be repeated. This property gives us termination but it also makes the equivalence proof harder.

The first implication turns out to be easy to prove since we are showing the existence of a particular form of derivation from a more generic one.

To prove the latter implication, we need to show that each derivation without any constraints on its form, can be recast into a derivation where the individual derivations of ϵ do not have repeated symbols. We do this by a complete induction on the length of the derivation and show that any derivation of the form $N \Rightarrow^* \epsilon$ can be recasted into a new derivation (possibly smaller), that gets accepted by $nullableML$.

This 'obvious' property of nullable derivations is usually 'assumed' in textbook proofs, but plays a centre role when proving the equivalence between a mathematical definition and an executable one.

With this equivalence we now know that execution of SML code will provide a behaviour corresponding to that of the formal HOL entity.

The executable `firstSet` and `followSet` definitions were defined in a similar way (by introducing a "seen" list in the computation). The termination and equivalence proof follow similar lines of reasoning.

An Executable `slrmac`. Another interesting termination case is encountered when we try to make `slrmac` definition executable. `slrmac` checks whether the resulting table for the grammar has any conflict or not. It is not strictly a necessary component of the parser generator but does assist in stating some of the proofs. For example, with this function we can assert that if we can build a parse table for a grammar and the input belongs in the language of the grammar, then the parser will output a parse tree.

Building the parse table involves traversing the state space to find the next state for each of the symbols in the grammar, starting from the initial state. `neighbours` takes a state and returns a state list. The state list contains states that can be reached by following each of the symbols in the input (*i.e.*, transitions one-level deep). It uses `symNeighbour` to shift the dot past the current symbol and get the state corresponding to it. The resulting state contains no duplicates (`rmDupes`). The condition `DISTINCT` ensures that we don't loop forever by considering states where the same items might be repeated. Another check, `validItl` makes sure that the items in the state do correspond to some rule in the grammar.

```
symNeighbour g itl sym =
         rmDupes (closure g (moveDot itl sym))

neighbours g itl [] = [] ∧
neighbours g itl (x::xs) =
  symNeighbour g itl x::neighbours g itl xs

visit g sn itl =
   if ¬(DISTINCT itl) ∨ ¬(validItl g itl) then []
   else let s = neighbours g itl set (allSyms g) in
     let rem = diff s sn in
           rem++(FLAT (MAP (visit g (sn++rem)) rem))
```

The parse table builder here is the `visit` function. Starting in the initial state it follows the transitions for each of the symbols in the grammar until it can reach no more new states. The important thing here is to make sure states are not repeated otherwise we end up following the same path over and over again. Here, the number of states seen increases at each recursive call. We also know that the number of possible states (even though it might be large) is finite (`allGrammarItls`). This is because we have a finite number of symbols in our grammar and a finite number of rules as well. From this we can deduce that the number of states that have not been encountered decreases at each call. This forms our termination argument.

```
measure (λ(g,sn,itl). |allGrammarItls g \ set sn|)
```

With this on hand, we can implement an executable `slrmac` that checks the entire table for shift-reduce and reduce-reduce conflicts.

```
slrML4Sym g [] sym = SOME (goto g, reduce g) ∧
slrML4Sym g (i::itl) sym =
    let s = goto g i sym in
      let r = reduce g i (sym2Str sym) in
        case (s,r) of ([],[]) -> slrML4Sym g itl sym
        || ([],[v12]) -> slrML4Sym g itl sym
        || ([],h::h'::t) -> NONE
        || (h::t,[]) -> slrML4Sym g itl sym
        || (h::t,h'::t') -> NONE

slrML g itl [] = SOME (goto g, reduce g) ∧
slrML g itl (sym::rst) =
    if (slrML4Sym g itl sym = NONE) then NONE
    else slrML g itl rst
```

4 Future Work

One piece of future work we would like to pursue is to demonstrate that SLR parsers terminate on all inputs, not just on strings in the language. This would then demonstrate the decidability of language membership. (Our mechanisation currently admits the possibility that `parser` goes into an infinite loop.)

We would also like to improve the efficiency of the parser. Currently, the DFA states are computed on the fly. This gives us simpler proof goals, assisting in reasoning about the program's properties. Changing this to be computed statically would enhance the performance of the parser when emitted as executable SML code.

For the sake of simplicity, we have dealt with SLR parsers. In practice however, compiler-compilers such as yacc and GNU bison generate LALR parsers. Instead of follow sets, LALR parsers uses lookahead sets, which are more specific as they take more of the parsing context into account, allowing finer distinctions. It will be interesting to see to what extent the existing work on SLR will assist us in verifying other parsing algorithms such as LR(1), LALR or GLR parser generator. We anticipate that most of the proof framework will not change excepting the work related to calculating lookahead sets.

5 Related Work

To realise the ambition of fully verified translation from source to machine code, all phases in the compilation process should either be verified or subject to verification after the fact. These two strategies are implemented in what have been termed *verified* or *verifying* compilers respectively. As we have already commented, one might imagine that the appropriate strategy for parsing would be to verify the output of an external tool. This then would be what one might call *verifying parsing*. For example, a verifying parser would mesh with Blazy, Dargaye and Leroy's work on the formal verification of a compiler front-end for a subset of the C language [1], which otherwise ignores parsing as an issue.

In the field of language theory, Nipkow [4] provided a verified and executable lexical analyzer generator. This is the closest in nature to the verification we have done. As with our work, Nipkow faced issues in making his definitions executable, principally because of the inductively defined transitive closure.

6 Conclusions

We have presented work on formal verification of an SLR parser generator. Most of the functions are directly executable. For those that we thought were better expressed more "mathematically", we have presented executable definitions of behaviourally equivalent alternatives. This conversion also illustrated the gap between simple textbook definitions and a verifiable executable implementation in a theorem prover. Issues like termination which can be ignored when dealing with semantic definitions, become necessary when executability comes into play. This also highlights how eminently suitable HOL is for developments of this kind, especially with its facility of emitting verified HOL definitions as SML code.

HOL sources for the work are available at `http://users.rsise.anu.edu.au/~aditi/`. The definitions and proofs are 21000 LOC. It took 7 months to complete the work which includes over 700 lemmas/theorems. This includes the definitions, major proofs related to SLR grammars and also lemmas about existing HOL types (*e.g.*, sets,lists) that were not already present in the system.

References

1. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 460–475. Springer, Heidelberg (2006)
2. Gordon, M.J.C., Melham, T. (eds.): Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, Cambridge (1993)
3. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
4. Nipkow, T.: Verified lexical analysis. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 1–15. Springer, Heidelberg (1998)
5. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008), `http://hol.sourceforge.net`