MICHAEL SPERBER
PETER THIEMANN

The essence of LR parsing

Originalbeitrag erschienen in:
Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation: La Jolla, California, June 21 - 23, 1995.
New York: ACM Press, 1995, S. 146-154

The Essence of LR Parsing

Michael Sperber Peter Thiemann
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13, D-72076 Tübingen, Germany

Abstract

Partial evaluation can turn a general parser into a parser generator. The generated parsers surpass those produced by traditional parser generators in speed and compactness. We use an inherently functional approach to implement general LR(k) parsers and specialize them using the partial evaluator Similix. The functional implementation of LR parsing allows for concise implementation of the algorithms themselves and requires only straightforward changes to achieve good specialization results. In contrast, a traditional, stack-based implementation of a general LR parser requires significant structural changes to make it amenable to satisfactory specialization.

1 Introduction

We present two inherently functional implementations of general LR(k) parsers: a direct-style first-order textbook version and one using continuation-passing style (CPS) for state transitions. Neither requires the handling of an explicit parsing stack.

These parsers, when specialized with respect to a grammar and lookahead k, yield efficient residual parsers. To achieve good results with offline partial evaluation, only a small number of changes to the general parsers are necessary:

- some standard binding-time improvements, notably some applications of "The Trick" [9] as well as some duplication of procedures which occur in multiple binding-time contexts,
- unrolling loops over lists to discard unneeded computations,
- prevention of infinite specialization of LR transitions for the CPS-based parser.

We describe the most important applications of the above improvements. The generated parsers are compact and are either about as fast or faster than those presented by Mossin [12]. His traditional stack-based parser requires substantial changes in the representation of the stack, and thus the structure of the parsing algorithm. Furthermore, since the parse stack is a static data structure under dynamic control, specialization suffers from termination problems. These issues do not arise in our first-order approach as it does not deal with explicit stacks at all. For the CPS approach, it is immediately obvious where generalization is necessary to prevent infinite specialization. Thus, the parsers modified for good specialization retain the structure of their ancestors and most of their simplicity.

The paper is organized as follows: the first section introduces the basic concepts of LR parsing along with a non-deterministic functional algorithm for it. Section 3 presents a deterministic, first-order, direct-style implementation of the algorithm in Scheme, and describes the binding-time improvements made to it. Section 4 describes an alternative formulation and Scheme implementation of functional LR parsing using CPS, again with a description of the binding-time improvements made to it. Section 5 describes some additional features which can be added to the parsing algorithms. Section 6 gives the results of practical experiments, Sec. 7 discusses related work, and Sec. 8 concludes.

2 LR Parsing

2.1 Notational Preliminaries

We use mainly standard notation for context-free grammars. However, the definition of bunches which follows is specific to the functional interpretation of LR parsing.

A context-free grammar is tuple $G = (N, T, P, S)$. N is the set of nonterminals, T the set of terminals, $S \in N$ the start symbol, $V = T \cup N$ the set of grammar symbols, and P the set of productions of the form $A \to \alpha$ for a nonterminal A and a sequence α of grammar symbols. Additionally, V^* denotes the set of sequences of grammar symbols—analogously T^* and N^*.

ϵ is the empty sequence, $|\xi|$ is the length of sequence ξ. Furthermore, α^k denotes a sequence of k copies of α, and $\xi_{|k}$ is the sequence consisting of the first k terminals in ξ.

Some letters are by default assumed to be elements of certain sets: $A, B, C, E \in N$; $\xi, \rho, \sigma, \tau \in T^*$; $x, y, z \in T$; $\alpha, \beta, \gamma, \delta, \nu, \mu \in V^*$, and $X, Y, Z \in V$. All grammar rules in the text are implicitly elements of P.

G induces the *derives relation* $\Rightarrow$ on V^* with

$$\alpha \Rightarrow \beta \;:\Leftrightarrow\; \alpha = \delta A \gamma \wedge \beta = \delta \mu \gamma \wedge A \to \mu$$

and $\overset{*}{\Rightarrow}$ denotes the reflexive and transitive closure of $\Rightarrow$. A *derivation* from α_0 to α_n is a sequence $\alpha_0, \alpha_1, \ldots, \alpha_n$ where $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \le i \le n$. *Leftmost-symbol rewriting* $\Rightarrow\!\!\!\!\!\Rightarrow$ is a relation defined as

$$B\alpha \Rightarrow\!\!\!\!\!\Rightarrow \delta\beta \;:\Leftrightarrow\; B \to \delta \wedge \delta \neq \epsilon$$

with reflexive and transitive closure $\overset{*}{\Rightarrow\!\!\!\!\!\Rightarrow}$.

An *LR(k) item* (or just item) is a triple consisting of a production, a position within its right-hand side, and a terminal string of length k—the *lookahead*. An item is written as $A \to \alpha \cdot \beta\ (\rho)$ where the dot indicates the position, and ρ is the lookahead. If the lookahead is not used (or $k = 0$), it is omitted.

An *LR(k)-augmented* grammar contains a terminal $\dashv$ denoting the end of the input, and a production $S' \to S\ \dashv^k$ where S' appears in no other production. Thus, it is start-separated, and all strings that can be derived from it end with $\dashv^k$. All grammars used are assumed to be LR(k)-augmented.

A *bunch* is a nondeterministic choice of values. If the a_i are bunches, $a_1|a_2|\ldots|a_k$ is a bunch consisting of the values of $a_1, \ldots, a_k$. An empty bunch is said to *fail* and therefore denoted by *fail*. In other words, | is the nondeterministic choice operator with unit *fail*. A bunch used as a boolean expression reads as false when it fails, and true in all other cases. Functions distribute over bunches. If a subexpression fails, the surrounding expressions fail as well. For bunches P and x, the expression $P \rhd a$ is a *guarded expression*: if the guard fails the entire expression fails; otherwise the value of $P \rhd a$ is a. For example, if P is a boolean expression (or a bunch), and x is a bunch, then $P \rhd x$ means **if** P **then** x **else** *fail*. The special guard $x \leftarrow a$ is a binding construct where $x \leftarrow a \rhd a'$ denotes the choice of a' over all values x "returned" by a (and fails if a fails). The scope of the variables in x (which may be a pattern) is exactly the right-hand a'.

2.2 Functional LR Parsing

A parser for a given context-free grammar is a function which has a sequence of terminals ξ as its input. It finds a derivation starting from the start symbol leading to ξ. The traditional model of an LR parser is an automaton which operates on a stack of parser states where every state is a set of items. When the automaton is in a certain state q, an item $A \to \alpha \cdot \beta \in q$ means that it has seen α in the input and is looking for β next. When $\beta = \epsilon$, the parser makes the decision to choose a derivation originating from A.

A functional LR parser is a set of mutually recursive functions [q], each of which corresponds to an LR state q. $[q](x_1 x_2 \ldots x_n)$ produces a bunch ranging over all possible derivations from β with an $A \to \alpha \cdot \beta \in q$ such that β derives a prefix of $x_1 x_2 \ldots x_n$. Each derivation found is represented by the item from which it originated and the rest of the input string not yet parsed. The LR parsing algorithm presented here adheres to the following specification (cf. [11]):

$$[q](x_1 x_2 \ldots x_n) = \begin{array}{l}(A \to \alpha \cdot \beta \in q \wedge \beta \stackrel{*}{\Rightarrow} x_1 \ldots x_j) \\ \quad \rhd (A \to \alpha \cdot \beta, x_{j+1} \ldots x_n)\end{array}$$

The initial state q_0 of an LR parser consists just of the item $S' \to \cdot S\ \dashv^k$. $S \stackrel{*}{\Rightarrow} x_1 \ldots x_n$ holds iff $(S' \to \cdot S, \epsilon)$ occurs in the bunch produced by $[q_0]\,(x_1 \ldots x_n)$.

A few auxiliary definitions are necessary to cast the definition into an algorithm. The $\mathtt{first}_k$ function (when constructing an LR(k) parser) is needed for computing lookahead information:

$$\mathtt{first}_k(\alpha) := \{\xi : |\xi| = k \wedge \alpha \stackrel{*}{\Rightarrow} \xi\beta\}$$

Each state q comes with a set of *predict items*

$$\mathtt{predict}(q) := \{B \to \cdot\nu\,(\rho) : \begin{array}{l}A \to \alpha \cdot \beta\,(\rho) \Downarrow^+ B \to \cdot\nu\,(\sigma) \\ \quad \text{for } A \to \alpha \cdot \beta\,(\rho) \in q\}\end{array}$$

where $\Downarrow^+$ is the transitive closure of the relation $\Downarrow$ defined by

$$A \to \alpha \cdot B\beta\,(\rho) \Downarrow B \to \cdot\delta\,(\sigma) \text{ for all } \sigma \in \mathtt{first}_k(\beta\rho).$$

The predict items are predictions on what derivations may be entered next. The elements of $\mathtt{predict}(q)$ are exactly those at the end of leftmost-symbol derivations starting from items in q. The union of q and $\mathtt{predict}(q)$ is called the *closure* of q. Henceforth, $\overline{q} := q \cup \mathtt{predict}(q)$ denotes the closure of a state q.

All other states result from applications of goto where, for a state q and a grammar symbol X:

$$\mathtt{goto}(q, X) := \{A \to \alpha X \cdot \beta\,(\rho) : A \to \alpha \cdot X\beta\,(\rho) \in \overline{q}\}$$

The general parser involves an auxiliary function $\overline{[q]}$ for each state q. An invocation $\overline{[q]}(X, x_1 \ldots x_n)$ means that a grammar symbol X, or a string derived from it has just been seen in the input. The specification is

$$\begin{array}{l}\overline{[q]}\ (X, x_1 \ldots x_n) = \\ \quad (A \to \alpha \cdot \beta \in q \wedge \beta \stackrel{*}{\Rightarrow} X\gamma \wedge \gamma \stackrel{*}{\Rightarrow} x_1 \ldots x_j) \\ \quad \rhd (A \to \alpha \cdot \beta, x_{j+1} \ldots x_n)\end{array}$$

Thus, $\overline{[q]}$ is called whenever the functional equivalent of a "shift" action happens. Its implementation calls $[\mathtt{goto}(q, X)]$ recursively and either returns the result (for a kernel item for the form $A \to \alpha \cdot \beta$ where $|\alpha| > 0$) or shifts the left side nonterminal of the production (for a predict item of the form $C \to \cdot\alpha$), this is just a recursive call of $\overline{[q]}$.

Figure 1 gives an "implementation" of the above functions detailing the algorithm underlying them based on the presentation in [11]. The function [q] first "shifts" on the next terminal symbol x. Additionally, if there is an ϵ-production $B \to \cdot$ in q, [q] shifts on B. Lastly, if q contains an item $A \to \alpha\cdot$ that can be reduced, it returns that item along with the part of the input string not yet seen. $\overline{[q]}$ first calls the function associated with $\mathtt{goto}(q, X)$ which returns an item and a "rest string". When the parser has arrived at a state containing an item $A \to \alpha\cdot$, it returns through $|\alpha| - 1$ function invocations, moving the dot back by one on each return: $A \to \alpha X \cdot \gamma$ returned by $[\mathtt{goto}(q, X)]$ becomes $A \to \alpha \cdot X\gamma$. The second alternative in $\overline{[q]}$ is considered when the dot has arrived at the front of a right-hand side $C \to \cdot X\delta$ by the process just described. In that case, $\overline{[q]}$ shifts on the corresponding left-hand side C.

Thus, the parsing stack of traditional LR parsers is implicit in the "runtime stack" of the procedure calls. Note also, that for returning through procedure invocations it is not really necessary to return the item itself: an integer counting the number of levels left to return through along with the left-hand side of the production is sufficient. Our implementation of the specification exploits this property.

3 Direct-Style Functional LR Parsing

A straightforward implementation of the LR (k) parsing algorithm is obtained by making the specification in

$$
\begin{aligned}
[q](x\xi) := \quad & \overline{[q]}(x,\xi) \\
\mid \quad & B \to \cdot\,(\rho) \in \mathtt{predict}(q) \wedge (x\xi)_{|k} = \rho \\
& \quad \triangleright\ \overline{[q]}(B, x\xi) \\
\mid \quad & A \to \alpha \cdot\ (\rho) \in q \wedge (x\xi)_{|k} = \rho \\
& \quad \triangleright\ (A \to \alpha\cdot, x\xi)
\end{aligned}
$$

$$
\begin{aligned}
\overline{[q]}(X,\xi) := \quad & A \to \alpha \cdot X\gamma \in q \wedge \\
& (A \to \alpha X \cdot \gamma, \tau) \leftarrow [\mathtt{goto}(q,X)]\,(\xi)\ \triangleright\ (A \to \alpha \cdot X\gamma, \tau) \\
\mid \quad & C \to \cdot X\delta \in \mathtt{predict}(q) \wedge \\
& (C \to X \cdot \delta, \tau) \leftarrow [\mathtt{goto}(q,X)]\,(\xi)\ \triangleright\ \overline{[q]}(C,\tau)
\end{aligned}
$$

Figure 1: Specification of a functional first-order LR parser

Fig. 1 deterministic. The $[q]$ function is implemented by a function `parser` which accepts a set of items and the input list. It produces a "parse result" which is one of `(result-accept)` and `(result-reduce lhs dot inp)`. The latter describes a pair consisting of an item `lhs` $\to \alpha \cdot \beta$ where `dot` $= |\alpha|$ and the remaining input `inp`. The `actions-on-input` function returns only those actions which apply to the current lookahead. Since there is no conflict detection there may be more than one reduce action in that list. The implementation arbitrarily chooses the first action.

Parse results are subsequently processed by the function `act-on`, the implementation of $\overline{[q]}$. The main difference is that `act-on` gets the result of calling $[\mathtt{goto}(q,X)](\xi)$ as an argument in place of the pair (X,ξ) and that the state q is implicit since the definition is nested in `parser`. The `(result-reduce lhs dot inp)` stands for a partially performed reduction of a production with left side nonterminal `lhs`, the position `dot` of the dot in the corresponding item, and the remaining input `inp`. The function `act-on` catches the results of `parser`. On the argument `(result-reduce` A r `inp)` there are two possibilities. If $r > 0$ more symbols must be popped: in this case `act-on` (and hence the surrounding `parse`) terminates with `(result-reduce` A $(r-1)$ `inp)`. Otherwise if $r = 0$ all symbols of the right-hand side of the reduced production have been popped and `parser` is called on the current state shifted by A and `inp`. The result is again `act-on`'d.

Notice that `act-on` is only called in tail-call position; in an imperative language, it might be implemented with a **while**-loop. One special case must be observed when shifting nonterminals: an attempt to shift the new start symbol S' in the initial state leads to `(result-accept)` if the input is exhausted and generates an error otherwise.

3.1 Specializing the Direct-Style Parser

Apart from two places where "The Trick" is needed, the direct-style parser specializes surprisingly well. There are no termination problems with the specialization, no explicit generalizations need be inserted due to the absence of an explicit stack in the functional parser and its first-order nature.

3.1.1 The Trick

"The Trick" is a well-known binding-time improvement [9]. It can be applied if a dynamic value d is known to belong to a finite set F of static values. The context $C[d]$ is replaced by a loop

foreach $s \in F$ **do**
$\{$ **if** $s = d$ **then** $C[s]$ **else continue** $\}$

which residualizes nicely to a cascade of tests where each occurrence $C[s]$ is as static as possible and can be reduced by the specializer.

Such a situation appears in two places in the parser: first in `actions-on-input`, when selecting the applicable actions based on the current state and the input, and second in `shift-items-on-symbol`, when shifting a nonterminal after a reduction is completed. In both cases we have introduced association lists of pairs consisting of a lookahead and a set of actions. In the first case the lookahead is a string of length k while it is just a nonterminal symbol in the second case.

Checking the lookahead on the input is done in such a way that the residual program does not contain superfluous operations on the dynamic input list and expands into a trie-like test cascade for $k > 1$. The very nature of the loop through the association list ensures the effect of grouping the default actions at the end of the tests. Section 4 shows the corresponding code in the CPS-based parser along with residual code generated by the specializer. The solution in [12] requires a programming trick on a CPS-version of the loop to achieve the same effect.

The static loop to select the nonterminal shift-actions (in `shift-items-on-symbol`) has a special property, by construction of the `goto` table: one of the tests in the loop is guaranteed to succeed, since a nonterminal shift on A is only performed after the reduction of a production $A \to \alpha$. Hence the last alternative to shift a nonterminal can be chosen without looking at the nonterminal A.

The latter improvement requires reprogramming the loop: the termination condition is modified so that the loop stops at the one-element list and simply performs the shift without testing. It is unlikely that an offline partial evaluator could perform that automatically since it would have to infer the above property. However, the necessary information could be provided by user annotations.

```
(define (parser items inp)
  (let ((state (closure items)))

    (define (act-on result)
      (let ((the-dot (rr-dot result))
            (the-lhs (rr-lhs result))
            (the-inp (rr-inp result)))
        (if (not (zero? the-dot))
            ; still symbols in front of the dot: continue popping
            (result-reduce the-lhs (- the-dot 1) the-inp)
            ; last symbol of rhs has just been popped:
            ; shift corresponding lhs and act-on goto (state, lhs)
            (if (zero? the-lhs) ;reducing the start production?
                (if (equal? (car the-inp) 'END-OF-INPUT)
                    (result-accept)
                    (err 'act-on "EOF expected"))
                (act-on (parser (shift-items-on-symbol the-lhs state)))))))

    (if (null? state)
        (err 'parser "Illegal input ~s" inp)
        (let* ((the-actions (actions-on-input inp state))
               (s-items     (get-shift-items the-actions))
               (r-actions   (get-reduce-actions the-actions)))
          (if (null? r-actions)
              (act-on (parser s-items (cdr inp)))
              (let ((r-action (car r-actions)))
                (if (null? (get-rhs r-action)) ;  epsilon production?
                    (act-on (parser (shift-items-on-symbol (get-lhs r-action) state) inp))
                    (result-reduce (get-lhs r-action)
                                   (- (length (get-rhs r-action)) 1)
                                   inp))))))))
```

Figure 2: Scheme implementation of direct-style functional parser

3.1.2 Improvements due to Properties of LR Parsers

Furthermore we included an improvement that was already observed by Mossin [12]: a state without items of the form $A \rightarrow \cdot\alpha$ does not have any shift actions on nonterminals. The nonterminal/action association list need not be checked in such a state since it is empty.

The check of the end of the input is only necessary in the initial state of the parser, *e.g.* the closure of $S' \rightarrow \cdot S$. Inserting an explicit test for that state in the code prevents the specializer from duplicating the test in all versions of `act-on`.

Finally, the initial state can never return a `result-reduce` since all its items have the dot at the beginning of their right-hand sides. That removes another test in the residual code.

3.1.3 Constructed Data

We modeled the parse results as constructed datatypes with the special forms provided for that purpose by Similix. As it turns out, they are partially static for some call sites of `act-on` but unfortunately this only gets apparent *after* specialization. Thus the argument to `act-on` must be classified as dynamic by any binding-time analysis. As a consequence, the residual code contains fragments like the following:

```
(let ((result_8
       (result-reduce 2 2 (cdr g-the-inp-4_2))))
  (act-on-1-19 (rr-lhs result_8)
               (rr-dot result_8)
               (rr-inp result_8)))
```

where the `rr-`*xxx* functions are the selector functions for the constructor `result-reduce`. The reduction of expressions like the one shown above can be achieved by making Similix's postprocessing phase somewhat more aggressive. Care must be taken, however, so as not to discard, duplicate, or reorder computations.

The newly introduced rule unfolds `(let ((x` e_1`)) `e_2`)` if e_1 is a constructor application `(C` v_1 ... v_k`)` for values $v_1, \ldots, v_k$ (a value is either a variable or a constant, a constant is either a literal constant or an evaluation-order-independent primitive operator applied to constants) and all occurrences of `x` in e_2 are of the form `(C-i x)` (a selector) or `(C? x)` (a constructor test).

Using the modified postprocessor effectively removed all occurrences of the above pattern. However, in some cases the pattern persisted since the constructor application was hidden in a function application where the whole body was a constructor application:

```
(define (tca-loop-0-13 one-reduce_0)
  (result-reduce 1 0 one-reduce_0))
```

A further tweak in the postprocessor removes these function calls as well: all functions whose bodies are just an application of a constructor, selector, constructor test, or primitive operation are unfolded (in addition to those that have only one call-site).

As an aside, note that the reductions performed by the postprocessor would have been performed by an online specializer anyway [17, 14].

Notice that duplication of code or using a polyvariant binding-time analysis would not improve the situation. The residual code shown above is generated from function

```
 (define (parser items inp)
   (let ((state (closure items)))

→    (define (act-on the-dot)
→      (let ((the-lhs (readvar-lhs))
→            (the-inp (readvar-inp)))
         (if (not (zero? the-dot))
             ; still symbols in front of the dot: continue popping
→            (- the-dot 1)
             ; last symbol of rhs has just been popped:
             ; shift corresponding lhs and act-on goto (state, lhs)
             (if (zero? the-lhs) ;reducing the start production?
                 (if (equal? (car the-inp) 'END-OF-INPUT)
                     'accept
                     (err 'act-on "EOF expected"))
                 (act-on (parser (shift-items-on-symbol the-lhs state)))))))

     (if (null? state)
         (err 'parser "Illegal input ~s" inp)
         (let* ((the-actions (actions-on-input inp state))
                (s-items     (get-shift-items the-actions))
                (r-actions   (get-reduce-actions the-actions)))
           (if (null? r-actions)
               (act-on (parser s-items (cdr inp)))
               (let ((r-action (car r-actions)))
                 (if (null? (get-rhs r-action))
                     ; reducing epsilon production: shift the lhs
                     (act-on (parser (shift-items-on-symbol (get-lhs r-action) state) inp))
→                     (begin
→                       (set-lhs! (get-lhs r-action))
→                       (set-input! inp)
→                       (- (length (get-rhs r-action)) 1)))
```

Figure 3: Direct-style parser using imperative features

applications `(act-on (parser ...))` in the source program. Since `(parser ...)` is a dynamic function call, its binding-time is always dynamic, not partially static. This is actually a well-known criticism on offline partial evaluators [14]. The residual function call is only unfolded by the postprocessor (if the specific specialized version is called only once) and the redex only arises at postprocessing time.

Our experiments revealed the constructed values to be a major performance bottleneck. The runtime of the residual parsers improves by a factor of about 2 just by changing the representation of the constructor `result-reduce` to a vector.

3.1.4 What You Get

All computations concerned with computing lookahead sets, items, closures, and actions are classified as static, as desired. Only direct dependencies on the input and some parse results remain dynamic. The partial evaluator performs the equivalent of the standard sets-of items construction on the fly using its caching mechanism. Furthermore, it performs an optimization on parse tables which is well-known in the compiler community [3, Chap. 6.3]: parse tables can be simplified by merging reduce actions with the immediately preceding shift action to a shift/reduce action if no conflicts can arise.

The residual code contains some duplicated code for the construction of results. However, experiments with inserting memoization points to increase sharing of code have not shown significant benefits.

3.2 Using Impure Features

Since the major bottleneck in the direct-style implementation results from the necessity to deliver multiple results from a function, we have looked at alternatives. One alternative, the transformation into CPS, is discussed in the next section since it has other intriguing features.

The other alternative is using impure features, i.e. side effects to global variables. It can be observed that only one `(result-reduce ...)` is live at any given time during the execution. Therefore, it is safe to replace its components by a couple of global variables. We choose to use global variables for the left-hand side nonterminal `lhs` and the remaining input string `inp` and pass the position of the dot `dot` in the right-hand side of the item as result. This is a reasonable choice since `dot` is subject to change with every termination of an invocation of `parser`. Figure 3 shows its implementation which is a simple variation of the code in Fig. 2 (the modified lines are indicated with →). The functions `set-lhs!` and `set-input!` set the respective global variables to their argument values. Notice that this is the only place in the program where the global variables are changed.

4 Continuation-Based LR Parsing

The algorithm for LR parsing in Sec. 2.2 can be converted into a more concise formulation using continuation-

```
(define (cps-parse grammar state input continuations)
  (if (and (final? state grammar)
           (null? input))
      'accept
      (let ((closure (append state (predict state grammar))))

        (define (c0 symbol input)
          (let ((next-state (goto closure symbol grammar)))
            (cps-parse
             grammar next-state input
             (cons c0
                   (take (- (active next-state) 1) continuations)))))

        (let ((item-set (accept closure)))
          (if (not (null? item-set))
              (let ((item (car item-set)))
                ((list-ref (cons c0 continuations)
                           (length (item-rhs item)))
                 (item-lhs item) input))
              (if (member (car input) (nextterm closure grammar))
                  (c0 (car input) (cdr input))
                  'error))))))
```

Figure 4: CPS-based functional parser

passing style [15]. The CPS-based parser is also amenable to good specialization. Note that the conversion to CPS is motivated by the desire to simplify the formulation of the parsing algorithm. It is not achieved through direct CPS transformation, and does not constitute a binding-time improvement by itself as is possible in many other contexts [9]. However, it allows for further binding-time improvements that do not apply to the direct-style version.

When the functional parser in Sec. 2.2 reduces by a production $A \to \alpha$, it has to return through $|\alpha|$ invocations of LR states—step by step. Continuations can be used to reach the correct state at once. The CPS-based parser needs a few new definitions. Each state q has an associated *number of active symbols* $active(q)$:

$$active(q) := \max(\{|\alpha| : A \to \alpha \cdot \beta \in q\} \cup \{0\})$$

When the parser is in state q, $active(q)$ is the maximal number of states through which it must return when it reduces by a production in q.

For formulating the CPS-based LR parser, it is convenient to make shifting on terminals subject to a condition. The set $nextterm(q)$ contains all terminals on which the LR state q might perform a shift action:

$$nextterm(q) := \{x : A \to \alpha \cdot x\beta \in \overline{q}\}$$

With these definitions, the CPS version of the functional parser is considerably simpler than the original: when a reduction occurs, the CPS version directly calls the continuation belonging to the correct state instead of shifting the dot back through multiple procedure invocations:

$$[q]\,(x\xi, c_1, \ldots, c_{active(q)}) :=$$
$$\quad \textbf{letrec}$$
$$\qquad c_0(X, \xi) =$$
$$\qquad\quad [goto(q, X)]$$
$$\qquad\qquad (\xi, c_0, c_1, \ldots, c_{active(goto(q,X))-1})$$
$$\quad \textbf{in} \quad A \to \alpha \cdot\ (\rho) \in \overline{q} \wedge (x\xi)_{|k} = \rho \quad \triangleright\ c_{|\alpha|}(A, x\xi)$$
$$\quad\ \mid \quad x \in nextterm(q) \qquad\qquad\qquad \triangleright\ c_0(x, \xi)$$

Again, for purposes of implementation, we are mainly interested in deterministic parsers. The LR(0) transliteration of the above specification is shown in Fig. 4. The state `state` is represented by a list of items, `continuations` is a list containing the $c_1, \ldots, c_{active(state)}$ from the specification. The `(take l n)` function extracts the first `n` elements of the list `l`, `accept` just extracts all items of the form $A \to \alpha \cdot$ from the closure of the current state.

The checking for lookahead to obtain parsers for $k > 0$ can be achieved in the same way as for the direct implementation in Sec. 3: A loop tests each of the next k input terminals in sequence, and filters out those reduce items which do not match it successively. Figure 5 shows the relevant code: The `select-lookahead-items` function returns items in `item-set` whose lookahead matches the first k terminals of `input`. (The `filter` function takes a predicate and a list as arguments and returns a list of those elements which match the predicate.) The code takes care not to check any terminal of the input for the same lookahead twice, as the specialized version would do that as well.

4.1 Specializing the CPS-Based Parser

Just as with the parser in Sec. 3.1, a few improvements are necessary to the naïve implementation of the parser so that it specializes well. The most obvious improvements presented there apply to the CPS-based version as well such as checking for the end of input only when in the final accepting state. The CPS parser needs a few specific improvements, however:

4.1.1 The Trick

Just as with the parsing algorithm from Sec. 2.2, a number of binding-time improvements are immediately obvious. The parameter `state` becomes static when "The Trick" is used in the application of the `goto` function. Here, the monovariance of Similix's binding-time analysis [2] requires some code duplication: since `c0` is passed as a parameter to `next-state`, both of its parameters necessarily

```
(define (select-lookahead-items item-set input)
  (let loop ((item-set item-set) (pos 0) (input input))
    (if (= pos k)
        item-set
        (if (null? item-set)
            item-set
            (let ((t (car input)))
              (let loop-2 ((items item-set) (used-lookaheads '()))
                (if (null? items)
                    items
                    (let* ((item (car items))
                           (pos-lookahead (list-ref (item-lookahead item) pos)))
                      (if (member pos-lookahead used-lookaheads)
                          (loop-2 (cdr items) used-lookaheads)
                          (if (equal? t pos-lookahead)
                              (loop (filter (lambda (item)
                                              (equal?
                                               (list-ref (item-lookahead item) pos)
                                               pos-lookahead))
                                            item-set)
                                    (+ pos 1) (cdr input))
                              (loop-2 (cdr items) (cons pos-lookahead used-lookaheads))))))))))))
```

Figure 5: Checking for lookahead

become dynamic. Hence there are three copies of the code for `c0`: for shifts on terminals, for shifts on nonterminals, and for the reduction of ϵ-productions.

Also, the `select-lookahead-items` function is subjected to "The Trick" as its return values are elements of a static set. The loop introduced by "The Trick" in `c0` is amenable to the same optimization as the parser in Sec. 3.1: since `c0` (after its duplication) can only be called with a nonterminal as its first argument, the last case need not be checked.

4.1.2 Unrolling Loops over Lists

Unfortunately, the residual parsers still contain accesses to `continuations` using `list-ref` as well as explicit constructions using `cons` and `take`. Since the first argument to `take`, `(- (active next-state) 1)`, is static, it is beneficial to unfold the call. A canonical definition of `take` follows:

```
(define (take n inl)
  (if (zero? n)
      '()
      (cons (car inl)
            (take (- n 1) (cdr inl)))))
```

This implementation of `take` has one shortcoming: the last recursive call to `take` evaluates `(cdr inl)` even though its value is never needed. Similix, to preserve termination properties of the program (the call to `cdr` might fail—so Similix assumes), inserts superfluous calls to `(cdr inl)` into the residual program. Inserting an additional test removes the deficiency:

```
(define (take n inl)
  (cond ((zero? n) '())
        ((= n 1) (list (car inl)))
        (else (cons (car inl)
                    (take (- n 1) (cdr inl))))))
```

Similar improvements can be made to the loop in `select-lookahead-items` in Fig. 5 which touches one symbol of `input` too many.

4.1.3 Removing the List of Continuations

Using standard Scheme lists for storing the continuations introduces another problem specific to Similix. Since it treats `cons`, `car`, and `cdr` as primitive operations and not as constructors and selectors, storing the continuations in a list makes them immediately dynamic [2, 1]. Even the first element of the list

```
(list-ref (cons c0 continuations)
          (length (item-rhs item)))
```

cannot be extracted if `(length (item-rhs item))` is 0.

The first step in solving the problem is to replace `cons`, `car` and `cdr` by calls to specially defined constructors for partially static data which Similix provides for. In consequence, `list-ref` and `take` also need to be adapted to use the new primitives.

Now, Similix can unfold all references to elements of `continuations`. This, unfortunately, leads to infinite specialization since Similix tries to unfold all instances of a given continuation (that is, in all possible call sequences that lead to it). This problem can be solved by applying a version of the `collapse` operator [1] to all uses of `c0`. `collapse` stops the infinite specialization of the recursive occurrence of `c0` in its own definition by inserting a generalization operator (which is just a dynamic identity function) nested in η-redexes. The η-redexes serve to separate the binding times of the context from the generalization operator. This separation is a general problem in offline partial evaluators. Only recently an automatic solution to it has been proposed [4].

The presence of `collapse` allows the specialization to stop at the generalization point but to still generate specialized code with respect to `c0`. Now, Similix completely reduces all references to `continuations`, effectively removing all references to the newly-defined list constructors.

```
(define (cps-parse-0-104 input_0 continuations_1)
  (let ((t_2 (car input_0)))
    (cond ((equal? t_2 'quote)
           (let ((input_6 (cdr input_0)))
             (if (equal? (car input_6) 'lisp-s-exp)
                 (let ((input_11 (cdr input_6)))
                   (if (equal? (car input_11) 'r)
                       (let ((input_16 (cdr input_11)))
                         (if (equal? (car input_16) 'r)
                             (if (equal? (car (cdr input_16)) 'r)
                                 (continuations_1 'exp input_16)
                                 (_sim-error 'cps-parse "Syntax error"))
                             (_sim-error 'cps-parse "Syntax error")))
                       (_sim-error 'cps-parse "Syntax error")))
                 (_sim-error 'cps-parse "Syntax error"))))
          ((equal? t_2 'if)
           (let* ((c_24 (lambda (x_22 y_23) #f))
                  (input_25 (cdr input_0))
                  (t_26 (car input_25)))
             (cond ((equal? t_26 'vname)
                    (cps-parse-0-219 (cdr input_25) (eta-expand-2-d-0-119 continuations_1 c_24)))
                   ((equal? t_26 'l)
                    (cps-parse-0-233 (cdr input_25) (eta-expand-2-d-0-119 continuations_1 c_24)))
                   (else (_sim-error 'cps-parse "Syntax error")))))))
```

Figure 6: Residual code fragment of an LR(2) parser

This optimization is crucial for the performance of the residual parsers; speed is improved by a factor of 2 through it.

Figure 6 shows some residual code for an LR(2) parser which also illustrates how the loop from Fig. 5 expands into a cascade of `cond`/`if` expressions.

5 Additional Features

There are a few features that can be added to the parsers and parser generators without problems.

5.1 Conflict Resolution

Conflict resolution techniques like SLR(k) and LALR(k) can be implemented. Preliminary experiments with a general SLR(k) parser (written in direct-style) show the expected results: specialization times get significantly shorter than for the canonical parsers and the sizes of the residual parsers shrink dramatically (see Table 1 in the next section). This is, of course, the usual trade-off between canonical LR and SLR/LALR parsers.

Including LALR parsers in our discussion would somewhat break the spirit of our approach. The computation of the LALR(1) lookahead sets requires the presence of the canonical collection of LR(0) items, which in our approach is only known to the specializer. For a general parser suitable to partial evaluation two approaches seem feasible:

1. The relevant part of the LR(0) item set collection is constructed in every state where a lookahead for a reduce action is needed.
2. The entire LR(0) item set collection is computed once and for all at the beginning of the program.

The first alternative would lead to unacceptable parser generation times, while the second leads to a design which is almost like a traditional parser generator: the partial evaluator would only be employed to generate code from the general parser, the construction of the LR automaton itself is done "by hand."

5.2 Semantics Actions

Semantic actions have been added to the direct style parser at the cost of introducing an explicit attribute stack (which is classified as dynamic by binding-time analysis). Only a single synthetic attribute per symbol is currently allowed, but that is not a real restriction in a language with higher-order functions and structured datatypes. The semantic action for a production is defined by a Scheme expression with free variables $\mathtt{x}_1$ to $\mathtt{x}_n$ denoting the attribute values of the symbols of the right-hand side of the production. The action is evaluated when a `result-reduce` is created, the constructor simply gets an additional slot for the attribute value.

Similix does not provide for closures as part of its input because this would require a special encoding. This shortcoming severely limits the usefulness of Similix for the generation of attribute-evaluating parsers because the semantic actions cannot be represented by closures. Instead, Scheme source expressions are needed. Evaluation is done by an interpreter for a subset of Scheme (in our case the first-order subset Scheme0 [9]) included with the parser. The specializer merely copies the text of the semantic actions from the grammar to the residual program.

An alternative approach suitable for Similix would be to simply load the subject grammar and the semantic actions along with the parser. This would circumvent having to write an interpreter, but suffers from obvious drawbacks: The general parser and the subject grammar would not be independent of each other, and therefore the approach would preclude the generation of a parser generator.

5.3 Nondeterministic Parsing

Nondeterministic parsing can easily be added to the continuation-based parser and hence to a parser generator by supplying failure continuations. This has been implemented in MØRK [16].

6 Experimental Results

With both implementation models, the generated LR parsers compare well with those generated by traditionally built parser generators such as MØRK [16] as well as those produced by the partial evaluation of a stack-based implementation presented in [12].

The two implementation models have different merits: while the direct-style approach leads to more compact parsers, the CPS-based parsers are faster in most cases. Table 1 shows the sizes of the input grammars and the sizes of the generated parsers. The sizes are given in the number of cells used for the Scheme representations. The column "Mossin" gives numbers for parsers generated from Mossin's general parser, "LR" for the direct-style parsers, and "CPS-LR" for CPS-based parsers. In column "SLR" we give the sizes of the residual direct-style SLR(1) parsers. Furthermore, the column "MØRK" contains the size of a MØRK-generated parser. Note that it is a LALR(1) parser (and hence the underlying automaton is smaller) but also includes trivial semantic actions which MØRK inserts automatically. The example grammars are those used by Mossin: G_1 defines balanced parentheses, G_2 arithmetic expressions, and G_3 the language Mixwell from [10]. We have used Similix 5.0 [1] for all our experiments.

Mossin	LR	CPS-LR
5575	3494	5196

Table 2: Size of the parser generators (in cons-cells)

For the parser generators built by applying Similix' compiler generator to the general LR(k) parser, similar results hold (cf. Table 2): for the direct-style approach, it is significantly shorter than those generated from Mossin's parser and from the CPS-based one. Notice that our parsers are both general LR(k) parsers whereas Mossin's does LR(1) parsing.

Table 3 shows the speed of the generated parsers over different grammars and inputs of varying sizes. The measurements were taken using Bigloo 1.7, a Scheme compiler which generates C code, on an IBM RS/6000 model 320 with 24MB of real memory running AIX >3.2.5 with maximum optimization. The C code was compiled using the native C compiler, xlc 1.3. The last column shows timings for equivalent LALR(1) parsers in C generated by Bison 1.22 and compiled with maximum optimization. The input for the Bison parser was fed directly from a constant array containing the token codes. Thus, all timings measure purely the parsing time.

The timings indicate that the speed of our functional parsers is within a factor of 2 of (with direct-style parsing) or surpasses (with the CPS approach) those generated from the stack-based approach by Mossin in [12]. The imperative direct-style version even surpasses those timings, and gets very close to the Bison-generated parsers in speed. These results prove the practicability of our approach. Earlier comparisons with MØRK using Scheme 48 indicate that the MØRK-generated parsers run about as fast as the CPS-based ones resulting from specialization—hardly a surprise since they are both derived from the same specification.

We also compiled a parser generator resulting from self-applying Similix with respect to the general parsers. For instance, in the LR(1) case, generation time for a CPS-based parser for G_3 was 22 seconds. Generation time significantly decreases for the SLR(1) case, however. It further decreases when the generator is run on a state-of-the-art workstation; another speed-up factor of four can be gained by using a PowerPC 250 instead of the somewhat outdated 320. Since it is possible to test a given grammar with the general parser, it is expected that the parser generator will have to be applied less often than is the case with Bison-generated parsers, for instance. Finally, it should be noted that the functions for item manipulation are not written with speed in mind. Using appropriate techniques [5] would result in further speed-up. Therefore, even the time to generate the parsers is sufficient for practical application.

7 Related Work

Of course, the pioneering work on functional LR parsing is due to Pennello [13]. He gives a low-level implementation of a direct-style functional parser.

An overview of functional parsing can be found in the Leermakers' book [11]. A summary along with a description of the CPS-based approach is given by the first author [15]. He uses the CPS-based parsing algorithm to implement sophisticated attribute evaluation. The resulting algorithms are used in the parser generator of the MØRK system for preprocessor generation [16].

As already mentioned in the paper, in [12], Mossin uses Similix to obtain specialized LR(1) parsers from general parsers starting with a stack-based first-order approach which does not specialize well. He transforms the stack data structure into a continuation which pops elements off the stack. The transformation complicates the program considerably and requires intricate binding-time improvements and other optimizations to achieve good specialization.

Another previous attempt to specialize general parsers is described in [6].

8 Conclusion

We have used the automatic specializer Similix to generate fast and compact LR(k) parsers from a general functional parser and a reference grammar. We used two alternative approaches to implement the general parser: one which represents the parsing stack by the procedure call stack, the other using continuation-passing style. No sacrifice of generality is necessary to make the programs amenable to good specialization. There is no need to cater to specific optimizations used in parser generators, or to $k = 1$.

Whereas a stack-based implementation of the parsing algorithms requires a quite intricate transformation of the stack data structure into a continuation to make it specialize well, the functional approach needs only straightforward and well-known improvements to generate good residual code.

G	Size (G)	Mossin	LR	CPS-LR	SLR	MØRK
G_1	24	1608	652	1236	491	
G_2	48	3751	2070	2999	1197	
G_3	123	6181	5870	7294	2700	6106

Table 1: Size of the residual parsers for k = 1 (in `cons`-cells)

G	Size	Mossin	LR	CPS-LR	LR-Imp	Bison
G_1	2	0.0637	0.0986	0.0480	0.0439	0.0220
	8	0.2030	0.2465	0.1343	0.0747	0.0586
	28	0.6376	0.7474	0.4470	0.1896	0.1785
G_2	3	0.1372	0.1732	0.0840	0.0554	0.0379
	13	0.4213	0.6273	0.3689	0.1254	0.1123
	35	0.9957	1.3755	0.9294	0.2605	0.2600
G_3	9	0.1293	0.3509	0.1211	0.0671	0.0600
	51	0.7699	1.6911	0.7523	0.3012	0.2847
	186	2.6190	5.8680	2.9020	1.0130	0.9470
	983	12.4240	31.0130	13.4630	4.5970	4.6280

Table 3: Speed of the residual parsers (timings in ms)

The generated parsers are faster and more compact than those generated by a straightforward parser generator. Together with our observation that parsers generated by partial evaluation compare well with *yacc*-generated parsers, we think that our results are quite encouraging towards a realistic application of partial evaluation to deliver production-quality parser generators. Work is planned to integrate our approach with the parser generator of the MØRK system and compare it with the previous implementation.

Availability The sources to the programs described in this paper are available via the World Wide Web in URL `http://www-pu.informatik.uni-tuebingen.de/users/sperber/lr-essence/`.

Acknowledgements We would like to thank Christian Mossin for graciously supplying us with the source code of his programs and sample inputs. Thanks are also due to the PEPM referees who provided detailed and constructive comments.

References

[1] BONDORF, A. *Similix 5.0 Manual.* DIKU, University of Copenhagen, May 1993.

[2] BONDORF, A., AND JØRGENSEN, J. Efficient analysis for realistic off-line partial evaluation. *Journal of Functional Programming 3*, 3 (July 1993), 315–346.

[3] CHAPMAN, N. P. *LR parsing: theory and practice.* Cambridge University Press, Cambridge, 1987.

[4] DANVY, O., MALMKJÆR, K., AND PALSBERG, J. The essence of eta-expansion in partial evaluation. In *Workshop Partial Evaluation and Semantics-Based Program Manipulation '94* (Orlando, Fla., June 1994), P. Sestoft and H. Søndergaard, Eds., ACM, pp. 11–20.

[5] DEREMER, F., AND PENNELLO, T. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. Program. Lang. Syst. 4*, 4 (Oct. 1982), 615–649.

[6] DYBKJÆR, H. Parsers and partial evaluation: An experiment. Student Report 85-7-15, DIKU, University of Copenhagen, Denmark, July 1985.

[7] IEEE. *Standard for the Scheme programming language.* Institute of Electrical and Electronic Engineers, Inc., 1991. Tech. Rep. 1178-1990.

[8] JOHNSON, S. C. Yacc—yet another compiler compiler. Tech. Rep. 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[9] JONES, N. D., GOMARD, C. K., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[10] JONES, N. D., SESTOFT, P., AND SØNDERGAARD, H. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications* (Dijon, France, 1985), J.-P. Jouannaud, Ed., Springer-Verlag, pp. 124–140. LNCS 202.

[11] LEERMAKERS, R. *The Functional Treatment of Parsing.* Kluwer Academic Publishers, Boston, 1993.

[12] MOSSIN, C. Partial evaluation of general parsers. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation '93* (Copenhagen, Denmark, June 1993), D. Schmidt, Ed., pp. 13–21.

[13] PENNELLO, T. Very fast LR parsing. *SIGPLAN Notices 21*, 7 (1986), 145–151.

[14] RUF, E. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, Stanford, CA 94305-4055, Mar. 1993. Technical report CSL-TR-93-563.

[15] SPERBER, M. Attribute-directed functional LR parsing. (submitted), Oct. 1994.

[16] SPERBER, M. MØRK: a generator for preprocessors. Master's thesis, Universität Tübingen, Mar. 1994. Available through URL `http://www-pu.informatik.uni-tuebingen.de/users/sperber/mork.ps.gz`.

[17] WEISE, D., CONYBEARE, R., RUF, E., AND SELIGMAN, S. Automatic online partial evaluation. In *Proc. Functional Programming Languages and Computer Architecture 1991* (Cambridge, MA, 1991), J. Hughes, Ed., Springer-Verlag, pp. 165–191. LNCS 523.