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, only a small number of changes to the general parsers are necessary:

- some binding-time improvements, notably some applications of "The Trick" 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 [9]. 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 value 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, Section 7 discusses related work, and Section 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 non-terminals, T the set of terminals, $S \in N$ is the start symbol, $V = T \cup N$ is the set of grammar symbols, and P is the set of productions of the form $A \rightarrow \alpha$ for a non-terminal 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. 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

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

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 $A \rightarrow \alpha \cdot \beta\ (\rho)$ 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 $S' \rightarrow S \dashv^k$ 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 an unordered sequence of values. If the a_i are "values", $a_1|a_2|\ldots|a_k$ is a bunch consisting of the values $a_1, \ldots, a_k$. An empty bunch is said to *fail* and therefore denoted by *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. The $\triangleright$ operator is used for conditional expressions: if P is a boolean expression (or a bunch), and x is a bunch, then $P \triangleright x$ means **if** P **then** x **else** *fail*. If x is a bunch, $a \leftarrow x$ means that a occurs in x.

2.2 Functional LR Parsing

This section contains a brief description of 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 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 $A \rightarrow \alpha \cdot \beta \in q$ 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. $[q](x_1x_2\dots x_n)$ 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 $x_1x_2\dots x_n$. 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. [8]):

$$[q](x_1x_2\dots x_n) = \left(A \to \alpha \cdot \beta \in q \wedge \beta \stackrel{*}{\Rightarrow} x_1 \dots x_j\right) \rhd (A \to \alpha \cdot \beta, x_{j+1} \dots x_n)$$

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

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:

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

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

$$\mathtt{predict}(q) := \left\{B \to \cdot\gamma\ (\rho) : A \to \alpha \cdot \beta\ (\rho) \Downarrow^+ B \to \cdot\gamma\ (\sigma) \text{ for } A \to \alpha \cdot \beta\ (\rho) \in q\right\}$$

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 $\mathtt{goto}$ 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 construction of a parser for the grammar includes an auxiliary function $\overline{[q]}$ for each state q. An invocation $\overline{[q]}(X, x_1 \dots x_n)$ means that a grammar symbol X, or a string derived from it has just been seen in the input. Thus, $\overline{[q]}$ is called whenever the functional equivalent of a "shift" action happens.

Here is an "implementation" of the above functions detailing the algorithm underlying them based on the presentation in [8]: 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 pass a whole item around: an integer representing 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.

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

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

3 A Direct Implementation of LR Parsing

A straightforward implementation of the LR (k) parsing algorithm is obtained by making the specification in Figure 1 deterministic. Conflict detection was omitted for simplicity's sake. 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 `actions-on` function returns only those actions which apply to the current lookahead.

Parse results are subsequently processed by the function `act-on`, the implementation of $\overline{[q]}$. The `(result-reduce ...)` stands for a partially performed reduction of a production with left hand non-terminal `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 non-terminals: 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 Parser

Apart from two places where "The Trick" is needed, the direct-style parser specialized surprisingly easy. 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.

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

    (define (act-on result)
      (let ((the-rhs (rr-rhs result))
            (the-lhs (rr-lhs result))
            (the-inp (rr-inp result)))
        (if (zero? the-rhs)
            ; last symbol of rhs has just been popped: 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))))
            ; otherwise (non-empty rhs) continue popping symbols
            (result-reduce the-lhs (- the-rhs 1) the-inp)))))

    (define (one-reduce r-action)
      (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)))

    (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)))
              (one-reduce (car r-actions)))))))
```

Figure 2: Scheme implementation of direct-style functional parser

3.1.1 The Trick

"The Trick" 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

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

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 non-terminal 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 non-terminal 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. Mossin [9] achieved this effect only by a programming trick on a CPS-version of the loop.

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

3.1.2 Improvements due to Properties of LR Parsers

Furthermore we included an improvement that was already observed by Mossin [9]: a state without items of the form $A \rightarrow \cdot\alpha$ does not have entries in the `goto` table. The non-terminal/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 result by the constructed datatypes as provided by Similix in the hope that they might turn out to be partially static in interesting places. Indeed, they are partially static for some call sites of `act-on` but unfortunately not for all of them. Thus the argument to `act-on` is classified as dynamic by Similix's monovariant binding-time analysis [2]. The places where the constructor application is partially static produce residual code as shown here:

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

where the `rr-xxx` functions are the selectors for `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 remained 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 on-line specializer anyway [13, 10].

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. Furthermore, the partial evaluator performs an optimization on parse tables which is well-known in the compiler community [3, Section 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.

4 Continuation-Based LR Parsing

The algorithm for LR parsing in Section 2.2 can be converted into a more concise formulation using continuation-passing style [11]. The CPS-based parser is also amenable to good specialization.

When the functional parser in Section 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* $active(q)$:

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

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 through multiple procedure invocations:

$$
\begin{array}{lll}
\multicolumn{3}{l}{[q]\ (x\xi, c_1, \ldots, c_{active(q)}) :=} \\
\quad \textbf{let} & \multicolumn{2}{l}{c_0(X, \rho) = [goto(q, X)]\ (\rho, c_0, c_1, \ldots, c_{active(goto(q,X))-1})} \\
\quad \textbf{in} & A \to \alpha \cdot\ (\rho) \in \overline{q} \wedge (x\xi)_{|k} = \rho & \rhd c_{|\alpha|}(A, x\xi) \mid \\
 & x \in nextterm(q) & \rhd c_0(x, \xi)
\end{array}
$$

Again, for purposes of implementation, we are mainly interested in deterministic parsers. The LR(0) transliteration of the above specification is shown in Fig. 3. The addition of lookahead information to obtain parsers for $k > 0$ can be achieved in the same way as for the direct implementation in Section 3.

The state `state` is represented by a list of items, `continuations` is a list containing the $c_1, \ldots, c_{active(\texttt{state})}$ 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.

```
(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 3: CPS-based functional parser

4.1 Specializing the CPS-Based Parser

Just as with the parser in Section 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 Section 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 become dynamic. Hence there are three copies of the code for `c0`: for shifts on terminals, for shifts on non-terminals, and for the reduction of ϵ-productions.

The loop introduced by "The Trick" in `c0` is amenable to the same optimization as the parser in Section 3.1: since `c0` (after its duplication) can only be called with a non-terminal as its first argument, the last case need not be checked.

4.1.2 Unrolling Loops over Lists

Unfortunately, the residual parsers still accesses `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)
  (let loop ((n n) (l '()) (inl inl))
```

```
    (if (zero? n)
        (reverse l)
        (loop (- n 1)
              (cons (car inl) l)
              (cdr inl)))))
```

This implementation of `take` has one shortcoming: the last call to `loop` in which `(reverse l)` is returned evaluates `(cdr inl)` even though it 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 whose results are never needed. Unrolling the loop removes the deficiency:

```
(define (take-2 n inl)
  (if (zero? n)
      '()
      (cons (car (cdr inl)) (take-2 (- n 1) (cdr inl)))))

(define (take n inl)
  (if (zero? n)
      '()
      (cons (car inl) (take-2 (- n 1) inl))))
```

Similar improvements could be made to the loop checking for correct lookahead (not shown here) which touched 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 nested in η-redexes. The η-redexes allow 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.

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

5 Additional Features

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

- Conflict resolution techniques like SLR(k) and LALR(k) can be implemented. Preliminary experiments with a general SLR(k) parser show that the specialization times get significantly shorter than for the canonical parsers. An implementation of the LALR conflict resolution would construct the canonical LR item closure and simplify it if no conflicts arise. This requires a slight restructuring of the parsers: the computation of the closure of the item set must take place at the call-site of the parser.

- 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). The semantic action for a production is a function which takes a list whose elements correspond to the attribute values of the symbols on 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. For obvious reasons, Similix does not allow functional objects in its input. Hence, actions are specified as quoted lambda abstractions which are passed to an interpreter for a subset of Scheme (a program can be supplied in addition to the grammar). The specializer was thus (mis-) used to copy the text of the semantic actions from the grammar to the residual program.

- Nondeterministic parsing can easily be added to the continuation-based parser and hence to a parser generator by supplying failure continuations.

6 Experimental Results

With both implementation models, the generated LR parsers compare well with those generated by traditionally built parser generators such as MØRK [12] as well as those produced by the partial evaluation of a stack-based implementations presented in [9].

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. The example grammars are those used by Mossin: G_1 defines balanced parentheses, G_2 arithmetic expressions, and G_3 the language Mixwell from [7]. We have used Similix 5.0 [1] for all our experiments.

G	Size	Mossin	LR	CPS-LR
G_1	24	1608	652	1236
G_2	48	3751	2070	2999
G_3	123	6181	5870	7294

Table 1: Size of the residual parsers for $k = 1$ (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.

Mossin	LR	CPS-LR
5575	3494	5196

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

G	Input size	Mossin 48	Mossin scm	LR 48	LR scm	CPS-LR 48	CPS-LR scm	MØRK 48
G_1	2	1.49	1.73	1.33	1.57	1.04	1.38	.
	8	3.98	4.50	3.53	4.41	3.27	3.47	.
	28	13.11	14.07	11.36	13.26	10.86	10.78	.
G_2	3	3.10	4.41	3.16	4.00	2.61	3.98	.
	13	9.93	10.85	10.43	11.93	8.64	9.38	.
	35	22.83	23.76	21.22	25.99	21.02	20.80	.
G_3	9	3.26	6.00	3.74	6.28	3.25	4.31	3.54
	51	17.26	22.19	19.28	21.31	17.41	14.20	19.83
	186	60.56	68.81	67.18	71.16	55.30	41.68	66.27
	983	298.18	208.61	361.41	367.58	306.02	191.89	.

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

Table 3 shows that the speed of the generated parsers is within a factor of 2 (with direct-style parsing) or surpasses (with the CPS approach) those generated from the stack-based approach by Mossin in [9]. The timings were taken using Scheme 48 0.39 with 4 Megawords of heap, and with scm 4e1, both on an IBM RS/6000 model 320 with 24MB of real memory running AIX 3.2.4. scm's garbage collection statistics also indicate that the CPS-based parsers allocate significantly less heap objects than those generated by Mossin's approach—sometimes by as much as a factor of 3.

Since Scheme 48 uses a byte-code engine, and *scm* is an interpreter, comparisons with parsers written in C or generated by *yacc* do not seem sensible. However, Mossin's measurements [9] indicate that a native-code Scheme compiler may indeed produce code that is slower "only" by a factor of 3–5 than that generated by *yacc*. In any case, the timings clearly indicate the practicability of the approach when used in conjunction with a high-performance Scheme system.

7 Related Work

An overview of functional parsing can be found in the Leermakers' book [8]. A summary along with a description of the CPS-based approach is given by the first author [11]. 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 which is described in [12].

As already mentioned in the paper, Mossin first used 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 which complicates the program and requires many intricate binding-time improvements and other optimizations to achieve good specialization.

A previous attempt to specialize general parsers is described in [4].

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 in the generality of the programs is necessary to make them 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.

Since the generated parsers are faster and more compact than those generated by a straightforward parser generator, it seems feasible to extend the approach to create a production-quality parser generator. 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 WWW as `http://www-pu.informatik.uni-tuebingen.de:/~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.

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

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

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

[7] 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.

[8] LEERMAKERS, R. *The functional treatment of parsing*. Natural Language Processing and Machine Translation. Kluwer Academic Publishers, Boston, 1993.

[9] MOSSIN, C. Partial evaluation of general parsers. In *SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (June 1993), D. Schmidt, Ed.

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

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

[12] SPERBER, M. MØRK: a generator for preprocessors. Master's thesis, Universität Tübingen, Mar. 1994. Available via WWW from `http://www-pu.informatik.uni-tuebingen.de/~sperber/mork.ps.gz`.

[13] 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.