Appeared in PEPM'93

Partial Evaluation of General Parsers

– Slightly Revised –

October 18, 1993

Christian Mossin*
DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø
Denmark
e-mail: mossin@diku.dk

Abstract

Applications of partial evaluation have so far mainly focused on generation of compilers from interpreters for programming languages. We partially evaluate a simple general LR(k) parsing algorithm. To obtain good results, we rewrite the algorithm using a number of binding-time improvements. The final LR(1) parser has been specialized using Similix, a partial evaluator for a higher order subset of Scheme [3]. The obtained specialized parsers are efficient and compact.

Partial evaluation is responsible for the (equivalent of) the *sets-of-items* construction, and the construction of parsing tables.

This paper is an extended abstract of [14].

1 Introduction

This paper describes an application of partial evaluation. We begin with a simple general LR(k) parser close to Knuth's original definition [12], and hope by partial evaluation to obtain specialized LR(k) parsers. Partial evaluation should be able to zcompute the set of LR(k) items, and the specialized parsers should contain (the equivalent of) *action* and *goto* tables, corresponding to LR parsers as presented in [1]. To obtain this, the program has to be (manually) transformed using a number of *binding-time improvements*. The specialized parsers obtained are fast and compact, and considering that they run in (untyped) Scheme, compares well to YACC.

In this work we use the partial evaluator Similix [3,5], which treats a higher order subset of Scheme with primitive operators and global variables. We use a notation for programs similar to, but more readable (we hope) than Scheme.

*This work was partly supported by ESPRIT Basic Research Actions project 3124 "Semantique"

1.1 Outline

The following section introduces partial evaluation, and section 3 recalls the standard definitions for LR parsing. Section 4 and section 5 defines two versions of our general LR(k) parser. The second version is improved in section 6. Section 7 describes the transformations we have done to improve the binding times of the parser. Section 8 shows the speedup gained by partial evaluation and compares our parser generator with YACC.

2 Partial Evaluation

Partial evaluation is a program transformation method for *specializing* programs. A partial evaluator *mix* applied to a program p and some of p's input (the *static* input) will yield a *residual* program p_{res}, which when applied to the remaining input of p (the *dynamic* input) will give the same result as p would, if applied to all of its input.

The partial evaluator Similix treats a higher order subset of Scheme with primitive operators and global variables. Similix is *polyvariant*, meaning that a function in the input program exists in several versions in the specialized program: one for each value of its static parameters.

2.1 Binding-time Improvement

Ideally the residual programs obtained by partial evaluation would be optimal, in the sense that all work that could be done on the basis of the static input has been performed. Even though partial evaluators have improved (and still will) this is of course not possible, since the evaluator cannot catch the intent of the program. Therefore the programmer has to do manual *binding-time improvements*, and thus transform the program in order to make larger parts static (executable at partial evaluation time).

2.2 Specializing a General Parser

A general parser *g-parser* is a parser taking both a grammar $\mathcal{G}$ and a string `ts` as input:

$$\textit{g-parser}(\mathcal{G},\texttt{ts}) = \textit{parsetree}$$

This can be viewed as a kind of (non-standard) interpreter, where $\mathcal{G}$ is the program to be interpreted, and the string `ts`

is its input. Similarly a specialized parser $parser_{\mathcal{G}}$ can be viewed as a (non-standard) compiled program taking strings as input. This leads to the following variant of the Futamura projections [2]:

1. $parser_{\mathcal{G}} = mix(g\text{-}parser, \mathcal{G})$
2. $parsergenerator = mix(mix, g\text{-}parser)$
   $= cogen(g\text{-}parser)$

where *parsergenerator* is a parser generator in the sense YACC is, and $cogen = mix(mix, mix)$.

3 Preliminaries

This section defines notation and auxiliary functions for our parser. Most of the definitions are standard and can be skipped at first reading, but are included for completeness.

Definition 3.1
Let $\mathcal{T}$ be the set of *terminals* (or *tokens*), let $\mathcal{N}$ be the set of *nonterminals*. We represent members of $\mathcal{T}$ by lowercase letters: *a, b, c* ..., and members of $\mathcal{N}$ by uppercase letters *A, B, C* ... . *Symbols* can be either terminals or nonterminals; this set is denoted $\mathcal{S} = \mathcal{T} \cup \mathcal{N}$ and *X,Y,Z* represents symbols. Strings of symbols are represented by greek letters: $\alpha, \beta, \gamma \ldots$. The empty string is ϵ. Further let $|\,\alpha\,|$ denote the number of symbols in α. We use α^k to denote $\overbrace{\alpha \cdots \alpha}^{k}$. A *production* has the form $A \rightarrow \alpha$. A *context free grammar* $\mathcal{G}$ is a set of productions with a designated nonterminal S called the *root*. That is, $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ where $\mathcal{P}$ is a set of productions.

We write $\phi \Rightarrow \psi$ with respect to grammar $\mathcal{G}$, if $\exists \alpha, \beta, \gamma$ such that $\phi = \alpha A \gamma$ and $\psi = \alpha\beta\gamma$ and $A \rightarrow \beta$ is a production in $\mathcal{G}$. We write $\alpha \stackrel{*}{\Rightarrow} \beta$ (β is derived from α) for the transitive and reflexive closure of $\Rightarrow$, that is if $\exists \alpha_0, \alpha_1, \cdots, \alpha_n$ $(n \geq 0)$ such that:

$$\alpha \ = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ \cdots \ \Rightarrow \alpha_n \ = \beta$$

A *sentential form* is a string α such that $S \stackrel{*}{\Rightarrow} \alpha$. A *sentence* is a sentential form consisting entirely of terminals. The *language defined by* $\mathcal{G}$, $L(\mathcal{G})$, is defined to be the set of sentences with respect to grammar $\mathcal{G}$. □

Definition 3.2
Let $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ be the grammar to be parsed by an LR(k) parser. Define the *augmented* grammar to be $\mathcal{G}' = (\mathcal{T} \cup \{\dashv\}, \mathcal{N} \cup S', (S' \rightarrow S \dashv^k) \cup \mathcal{P}, S')$. Add $\dashv^k$ to the end of the string `ts` to be parsed. □

Definition 3.3
$FIRST_k(\alpha)$ is defined to be the set of k-letter strings that can begin the strings derivable from α:

$$\begin{aligned}&\mathrm{FIRST}_k(\alpha) = \\ &\quad \{b_1 \cdots b_k \mid b_i \in \mathcal{T} \cup \{\dashv\}, 1 \leq i \leq k \wedge \\ &\qquad \exists \gamma : \alpha \stackrel{*}{\Rightarrow} b_1 \cdots b_k \gamma \ w.r.t.\ \mathcal{G}'\}\end{aligned}$$

□

Definition 3.4
An LR(k) *item* is a triple containing a production, a position within the right hand side of the production and a lookahead consisting of k terminals. The position is often written as a dot ($\cdot$) in the production, *e.g.* (T → l · r,r) is an LR(1) item. □

Definition 3.5
The $CLOSURE_k$ of a set of items I is defined recursively to be the smallest set I' satisfying the following equation:

$$\begin{aligned}I' = I \cup \{(A \rightarrow \cdot\, \alpha, \beta) \mid &\exists (B \rightarrow \gamma \cdot A\delta, \zeta) \in I' \wedge \\ &\beta \in \mathrm{FIRST}_k(\delta\zeta)\}\end{aligned}$$

□

The parser is always in some *state* (corresponding to a state in a DFA). A state corresponds to a set of items — in the following we will not make a distinction between the two. The initial state I_0 is $\mathrm{CLOSURE}_k\{(S' \rightarrow \cdot\, S, \dashv^k)\}$. Viewing states as the states of a DFA, the GOTO_k function of definition 3.6 corresponds to the transitions from state I on nonterminals X or strings $a_1 \ldots a_k$.

Definition 3.6
If I is a set of items, $X \in \mathcal{N}$ and $a_1 \ldots a_k \in \mathcal{T}^k$, define:

$$\begin{aligned}&\mathrm{GOTO}_k(I, a_1 \ldots a_k) = \\ &\quad \mathrm{CLOSURE}_k(\{(A \rightarrow \alpha a_1 \cdot \beta, \gamma) \mid \\ &\qquad (A \rightarrow \alpha \cdot a_1\beta, \gamma) \in I \\ &\qquad \wedge a_2 \ldots a_k \\ &\qquad\quad \in \mathrm{FIRST}_{k-1}(\beta\gamma)\}) \\ &\mathrm{GOTO}_k(I, X) = \\ &\quad \mathrm{CLOSURE}_k(\{(A \rightarrow \alpha X \cdot \beta, \gamma) \mid \\ &\qquad (A \rightarrow \alpha \cdot X\beta, \gamma) \in I\})\end{aligned}$$

□

The algorithms for computing FIRST, CLOSURE and GOTO are intensionally left unspecified, since these functions should be computed by the specializer, and will thus not appear in our specialized parsers.

4 First Order Version of the Parser

With the definitions of section 3, we are now able to define our general parser.

First we define two non-standard auxiliary functions. *shiftable* finds the items in state I on which a *shift* action on input $a_1 \ldots a_k$ is possible. Similarly, *reducible* finds the items in state I on which a *reduce* action on input β is possible. The idea is to discover possible conflicts, before doing the *shift/reduce*-action.

Definition 4.1
shiftable and *reducible* is defined for LR(k) parsers by:

$$\begin{aligned}&shiftable_k(I, a_1 \ldots a_k) = \\ &\quad \{(A \rightarrow \alpha \cdot a_1\beta, \gamma) \mid (A \rightarrow \alpha \cdot a_1\beta, \gamma) \in I \\ &\qquad \wedge a_2 \ldots a_k \\ &\qquad\quad \in \mathrm{FIRST}_{k-1}(\beta\gamma)\}\end{aligned}$$

$$\begin{aligned}&reducible(I, \beta) = \\ &\quad \{(A \rightarrow \alpha \cdot, \beta) \mid (A \rightarrow \alpha \cdot, \beta) \in I\}\end{aligned}$$

□

In figure 1, our LR(k) parser is defined in first order style. This was the parser, with which our work originally started. It is almost identical to the original definition by Knuth [12]. In the next section, the parser will be reformulated in a higher order style. The first order parser of figure 1, will be used for measuring speedups in section 8.

We use `stack` to denote the stack. We give $\mathcal{G}$ as a parameter to *parse*, but otherwise leave $\mathcal{G}$ and $\mathcal{G}'$ implicit. `r/r-` and `s/r-confl` means reduce/reduce- resp. shift/reduce-conflict. Conditionals can be written as "`cond...`" or "`if elsf elsf...`" (that I use both, is merely a matter of personal taste).

```
parse(G,ts) =
  parseloop(push(CLOSURE_k({(S'→·S,⊣^k)}),
                 stack_empty),
            ts++⊣^k)

parseloop(stack, t1···tk:ts) =
  let I = top(stack)
  in if I=∅ 'rejected
     elsf I={(S'→S⊣^k·,⊣^k)} 'accepted
     else
      let s-items = shiftable(I,t1···tk)
          r-items = reducible(I,t1···tk)
      in cond
          (s-items=∅ ∧ r-items={(A→α·,β)})
            reduce((A→α·,β),stack,t1···tk:ts)
          (s-items≠∅ ∧ r-items=∅)
            shift(I,stack,t1···tk:ts)
          (|r-items|≥2) 'r/r-confl
          (s-items≠∅ ∧ r-items≠∅) 's/r-confl
          (s-items=∅ ∧ r-items=∅) 'rejected

reduce((A→α·,β),stack,ts)
  let stack1 = pop(|α|,stack)
      stack_new = push(GOTO_k(top(stack1),A),stack1)
  in parseloop(stack_new,ts)

shift(I,stack,t1···tk:ts)
  let stack_new = push(GOTO_k(I,t1···tk),stack)
  in parseloop(stack_new,t2···tk:ts)
```

Figure 1: *First order Parser*

5 Higher Order Version of the Parser

This section redefines our general parser; since we use a higher order functional language, it is natural to use a higher order representation of the stack. Since programming in a higher order style seems to be a trend of the future, we believe our first choice of first order style was primarily due to tradition. Choosing higher order style, turned out to be crucial for succesful partial evaluation — in subsection 5.1 we will explain why.

First we notice that both *shift* and *reduce* actions will push a state onto the stack, and that this state will immediately be needed at the top of function *parseloop*. We can easily separate the state on top of stack from the underlying stack and make it a seperate parameter in *parseloop*.

This separation has another effect: states on the stack is now only used one place, namely in a *reduce* action. We can now use an idea described by Schmidt [16], where a command stack is turned into a function.

Let us first look at such a transformation in general terms. We present the transformation in two steps.

Assume a program working on a stack `stack` of $y_1, y_2, \ldots$. Assume that any call to *pop* is followed by a call to a function f, *i.e.* $f(pop(\texttt{stack}), x)$, where x is some variable(s). We cannot convert `stack` into a stack of $f(y_i)$, since f also depends on x, so instead we transform it into a stack of continuations $\lambda(x).f(y_i, x)$.

We can even represent the stack itself as a continuation, taking the number of elements to pop as argument. The new stack now has the structure $\lambda(n).\texttt{if } n = 1\ \lambda(x).f(y_1, x)\ \texttt{elsf } n = 2\ \texttt{then } \lambda(x).f(y_2, x)\cdots$.

Figure 2 shows the parsing algorithm. The continuation replacing the stack is called κ^{stack}. For building the continuation we keep the term *push*. *pop* will consist of applying the continuation. The continuation will first take a number of elements to pop. The result of this will be a new continuation κ^{state}, corresponding to the actions that would be performed on the state I on top of the stack. κ^{state} will consist of a call to *parseloop* with $\text{GOTO}_k(I,\texttt{nt})$ as the current state. κ^{state} thus needs `nt` and the current string `ts` as arguments. Further the call to *parseloop* need a continuation (a stack); this should be the same continuation as we are currently applying. Thus κ^{state} should have itself as final argument. The type of κ^{stack} and κ^{state} becomes:

$$\begin{array}{lcl} \kappa^{stack} & :: & \mathbb{N} \rightarrow \kappa^{state} \\ \kappa^{state} & :: & \mathcal{N} \times \mathcal{T}^* \times \kappa^{state} \rightarrow result \end{array}$$

Function *parse* is almost identical to the first order version; the initial state I_0 is not pushed on κ^{stack}_{empty}($\kappa^{stack}_{empty}$is defined by λ(n).λ(nt,`ts`,κ^{state}).$\bot$). Similarly, *parseloop* is only changed to take the separate parameter I into account.

Notice in the definition of *reduce*, reducing an empty production corresponds to a *push*. This is because we made the current state a separate parameter.

```
parse(G,ts) =
  parseloop(λ(n).λ(nt,ts,κ^state).⊥,
            CLOSURE_k({(S'→·S,⊣^k))},
            ts++⊣^k)

parseloop(κ^stack,I,t1···tk:ts) =
  if I=∅ 'rejected
  elsf I={(S'→S⊣^k·,⊣^k)} 'accepted
  else
   let s-items = shiftable(I,t1···tk)
       r-items = reducible(I,t1···tk)
   in cond
       (s-items=∅ ∧ r-items={(A→α·,β)})
         reduce((A→α·,β),I,κ^stack,t1···tk:ts)
       (s-items≠∅ ∧ r-items=∅)
         shift(I,κ^stack,t1···tk:ts)
       (|r-items|≥2) 'r/r-conflict
       (s-items≠∅ ∧ r-items≠∅) 's/r-conflict
       (s-items=∅ ∧ r-items=∅) 'rejected

reduce((A→α·,β),I,κ^stack,ts) =
  let to-pop = |α|-1
  in if to-pop≥0
       let κ^state = κ^stack(to-pop)
       in κ^state(A,ts,κ^state)
     else let κ^stack_new = push(I,κ^stack)
```

```
                    I_new=GOTO_k(I,A)
                in parseloop(κ_new^stack,I_new,ts)

shift(I,κ^stack,t_1 ··· t_k:ts) =
  let κ_new^stack = push(I,κ^stack)
      I_new = GOTO_k(I,t_1 ··· t_k)
  in parseloop(κ_new^stack,I_new,t_2 ··· t_k:ts)

push(I,κ^stack) =
  (λ(n).if n=0
        λ(nt,ts,κ_1^state).
           parseloop(λ(n).if n=0
                          λ(nt,ts,κ_2^state).
                           κ_1^state(nt,ts,κ_1^state)
                          else κ^stack(n-1),
                     GOTO_k(I,nt),
                     ts)
        else κ^stack(n-1))
```

Figure 2: *The stack represented by a continuation*

5.1 The Effect on Binding Times

As mentioned above, replacing the stack by a continuation, is crucial for obtaining good specialization. On the other hand, it is debatable whether or not it really is a binding time improvement, since it is a matter of style, whether a programmer would have started with the algorithm of figure 1 or the algorithm of figure 2.

It is often the case, that a continuation representing a stack, is better (more efficient), than a first order representation, even when partial evaluation is not involved. In partiel evaluation, it has been known for some time that this is a good idea — we will try to clarify this:

We will later (in subsection 7.1) see that the stack has to be made dynamic to ensure termination of specialization. If the stack is represented by a list $y_1, y_2, \ldots$, the elements will then become dynamic as well. If the stack is represented by a continuation, $\lambda(n).\texttt{if } n = 1\ \lambda(x).f(y_1, x)\ \texttt{elsf } n = 2\ \texttt{then}\ \lambda(x).f(y_2, x)\cdots$, the applications $f(y_i, x)$ can be reduced as much as possible. The intuition is that, by representing the stack as a continuation, we explicitly tell the specializer, that the same function will always be applied to the elements on stack, and the specializer can use this information to reduce more.

Using the terminology used to describe the transformation from a list representation to a functional representation, it is made possible for f to be applied to static instances of y_i and thus be reduced as much as possible even though y_i is an element of a static structure under dynamic control.

In our case, this will help us ensure that calls to *parseloop* will always be with a static current state I.

The first part of the transformation (from stack of states to stack of continuations) is by far the most important, the second (from stack to continuation) has some more subtle advantages, see subsection 7.1.

6 Improving the Algorithm

In this section we do a few simple optimizations on the algorithm of section 5. The optimizations improves the performance of the algorithm slightly, but will have greater impact when we specialize.

6.1 The top of the stack is not always needed

Sometimes we can avoid building some of the continuation, since we can predict, that it will not be used. The idea is to predict cases where the argument `n` to κ^{stack} cannot equal zero.

6.1.1 Building a Continuation using *push*

Reduction always lead to a state where we began reading the rightside of a production. Thus, if we know that a state does not contain any items of the form $(A\rightarrow\cdot\,\alpha,\beta)$, we will know that `n` cannot equal `0`. If this is the case we know that we just have to pass $n-1$ to the continuation on which we are building. A predicate for identifying these situation is defined in definition 6.1. This optimization of *push* together with the ones of the next subsubsection are presented in figure 3.

Definition 6.1
any-initial?$(I) = \exists A\alpha\gamma : ((A\rightarrow\cdot\,\alpha,\gamma) \in I)$ □

6.1.2 The continuation passed to the *parseloop* in *push*

The argument continuation to *parseloop* (see definition of *push* in figure 2):

```
λ(n).if n=0
      λ(nt,ts,κ_2^state).
       κ_1^state(nt,ts,κ_1^state)
      else κ^stack(n-1)
```

is only used, if the state we are pushing is used a second time. That is, first there must be a reduction on some $(A\rightarrow\cdot\,\alpha,\beta)$, then a reduction on some $(B\rightarrow\cdot\,\beta,\gamma)$ (not necessarily different from the first). We now know that the first symbol in β must be A. So we only have to build this continuation, if the state contains one or more items of the form $(B\rightarrow\cdot\,A\delta,\gamma)$. The predicate is defined in definition 6.2. The new algorithm for *push* can be seen in figure 3. The two places, where we avoid building a new continuation, are marked by a *. Otherwise it is identical to *push* of figure 2.

Definition 6.2
initial?$(I, A) = \exists B\beta\delta : ((B\rightarrow\cdot\,A\beta,\delta) \in I)$ □

```
push(I,κ^stack) =
  if ¬any-initial?(I)
     λ(n).κ^stack(n-1) *
  else
    λ(n).
     if n=0
        λ(nt,ts,κ_1^state).
         parseloop(if ¬initial?(I,nt)
                      λ(n).κ^stack(n-1) *
                   else
                     λ(n).if n=0
                           λ(nt,ts,κ_2^state).
                            κ_1^state(nt,ts,κ_1^state)
                           else κ^stack(n-1),
```

```
                GOTO_k(I,nt),
                ts)
    else κ^stack(n-1))
```

Figure 3: *Improved version of push*

6.2 The bottom of the stack is never needed

Due to the nature of the LR parsing algorithm, we know, that the empty stack will never be reached, so we would like to get rid of it. This can be done by treating the first "real" state as a special case when pushing onto the stack. The first real state is CLOSURE($\{(S' \rightarrow \cdot\, S, \dashv^k)\}$). This can be identified by `member?(`$(S' \rightarrow \cdot\, S, \dashv^k)$`,`$I$`)`. When pushing this state we know, that no further popping is needed — in other words, the argument n is zero. We also know that *any-initial?* will be true on this state, so we can omit the test.

This is not really an improvement; we have saved a test (`n=0`) in some cases, but at the price of a more complicated test (`member?(`$(S' \rightarrow \cdot\, S, \dashv^k)$`,`$I$`)`). Only when we look at binding times this a real advantage; we have in some cases replaced a dynamic test by a static one.

7 Binding-time Improvements

This section discusses how the algorithm as presented so far can be transformed, in order to obtain fast residual programs by partial evaluation.

In [6] Consel and Danvy describe how binding-time improvements on a naive pattern matching algorithm leads to a DFA recognizing a string as a substring of another string. One state of the DFA is represented by one function, and the transitions by calls. We would like the states and GOTO function of our parser to be represented in a similar way.

The idea is to exploit the polyvariancy and memoization of the specializer: if a parameter I is static, the residual program will contain one instance of the main-loop for each possible state, and calls between the different instances will represent transitions. To identify (set-) equal states as equal, we represent states as *sorted* lists of items.

The first thing to do however, is to avoid infinite specialization.

7.1 Avoiding Infinite Specialization

The stack is a static structure under dynamic control — the specializer will attempt to make residual function for each of the (infinitely many) static values, which the stack can evaluate to. This can be avoided using primitive function `generalize`. This function simply forces its argument to become dynamic.

We can avoid infinite specialization, without having to make the continuation dynamic. This is achieved using function `collapse`: Intuitively `collapse` "breaks" the infinite specialization where it is inserted, but allows straight-line code (without calls to `collapse`) in the continuation to be reduced. `collapse`[1] is described in the Similix Manual [4].

[1] `collapse` is defined by: is:

```
collaps(c) =
  eta-convert-s (generalize(eta-convert-d(c)))

eta-convert-d(c) λ(x).c(x)

eta-convert-s(c) λ(x).c(x)
```

Refer to the Similix Manual [4] for a description of how `collapse` works (it is called `generalize-c` in the manual).

This can be exploited on our continuation. When we actually build the continuation, the stack might grow infinitely. This happens when *any-initial?* is true and when *initial?* is true; these places we have to insert `collaps`. But if one of these is false, reductions can be done at specialization time without risk of infinite specialization, so in these places we do not have to use `collapse`.

7.2 Applying "the trick"

"The trick" (see *i.e.* section 4.8 in [8]) is a well known binding-time improvement, applicable in cases where a dynamic variable is known to belong to some statically computable finite set. The idea is then to look up the dynamic variable in the static set, and continue with the static version of the value instead of the dynamic. "The trick" can be implemented in a nice and general way using continuations:

```
c-member(x,y:ys,c) =
  if (ys=∅) ∨ (x=y)
     c(y)
     c-member(x,ys,c)
```

where `x` is the dynamic value known to belong to the statically known set `y:ys`. If we know $\mathtt{x} \in S$, and program piece $P(\mathtt{x})$ uses `x`, the program piece is transformed into `c-member(`$\mathtt{x}, S, \lambda(\mathtt{x}').P(\mathtt{x}')$`)`.

The `if` in the definition of `c-member` is dynamic, so when `c-member` is unfolded, we get a `cond`-statement in the residual program (= specialized parser). The test `(ys=`$\emptyset$`)` is a static test necessary to stop unfolding `c-member`. Note that we use (ys=$\emptyset$) instead of (y:ys=$\emptyset$), since we know that if the set `y:ys` contains exactly one element (`y`), `x` must equal `y`.

"The trick" can be used two places in our algorithm:

1. $\mathtt{t}_1 \cdots \mathtt{t}_k$ (the first tokens in the current string — third parameter of *parseloop*) is known to be a member of $\mathcal{T}^k$. We thus insert a call to `c-member` in the else branch (after `'accepted`):

   ```
   else
    c-member(t_1 ··· t_k, T^k,
             λt'_1 ··· t'_k
              let s-items = shiftable(I,t'_1 ··· t'_k)
              ...
   ```

2. `nt` (the argument of the first λ`(nt,ts,`κ_1^{state}`)` in *push*) is known to be a member of $\mathcal{N}$. We thus insert a call to `c-member` here:

   ```
   λ(nt,ts,κ_1^state).
    c-member
     (nt,N,
      λ(nt').
       parseloop(if ¬initial?(I,nt')
       ...
   ```

We are now able to specialize the algorithm and obtain the states in the way we wanted. The calls to `c-member` makes the residual program contain conditionals with branches for each terminal resp. nonterminal, corresponding to a lookup in a parse table.

7.2.1 An improvement of "the trick"

In the residual conditionals originating from `c-member`, many branches result immediately in a rejection. The specialized parser would be shorter and more readable (probably also a bit more efficient since Scheme goes through the branches of a conditional from one end), if we could have the "interesting" branches first, and all the rejecting branches collapsed into one `else`-branch.

In general the idea can be used, if some values in the static set S result in a default value v. If we can determine whether a value $s \in S$ results in v using a predicate P, we can rewrite `c-member` as:

```
c-member-improved(x,S,c) =
  if (S = ∅) v
  else
   let y:ys = S
   in if (¬P(y) ∧ (x=y)) c(y)
      else c-member-improved(x,ys,c)
```

If $P(x)$ holds the last test is statically known to be false, thus no residual `if` is generated. When S is empty, the default value v is returned. This becomes the else branch of the residual conditional. The benefit from using `c-member-improved` is illustrated by the following example: If S is $\{1,2,3,4,5\}$ and 'hello is returned for odd numbers ($P(x) =$ `odd`(x)), the residual conditional will change as follows:

```
cond                                  cond
  x=1 'hello                            x=2 ...
  x=2 ...                               x=4 ...
  x=3 'hello      ↪                     else 'hello
  x=4 ...
  x=5 'hello
```

In cases where no member of S results in v, the `else` branch can be avoided using a flag. How to do this is illustrated by our concrete use below. Here P will also need more (static) arguments than `x`.

On terminals

The immediate rejections in the branches of the conditional over terminals, stem from the test `(s-items = ∅ ∧ r-items = ∅)` at the end of function *parseloop*. But we can perform this test a bit earlier, namely on all the static members of $\mathcal{T}^k$. Hence we calculate `s-items` and `r-items` for the static values of $\mathtt{t}_1 \cdots \mathtt{t}_k \in \mathcal{T}^k$. If some terminals are rejected this way, we raise a flag `rej`, and stop the loop when we have tried all terminals returning `'rejected`. Otherwise (if the flag has not been raised) we can stop when one member of $\mathcal{T}^k$ is left as in `c-member` above.

```
c-member-t(t1···tk,t-set,I,rej,c) =
 if (t-set=∅) 'rejected
 else
  let (t1^stat,···,tk^stat):ts= t-set
  in if ((ts= ∅) ∧ ¬rej) c(t1^stat···tk^stat)
  else
   let s-items = shiftable(I,t1^stat···tk^stat)
       r-items = reducible(I,t1^stat···tk^stat)
   in if ((s-items≠ ∅) ∨ (r-items≠ ∅))
                  ∧ (t1···tk=t1^stat···tk^stat)
       c(t-stat)
  else c-member-t(t1···tk,ts,I,
                  ((s-items = ∅) ∧ (r-items = ∅)
                   ∨ rej),
                  c)
```

Figure 4: *"The trick" for terminals*

On nonterminals

The immediate rejections in the conditional over nonterminals come from the test at the top of function *parseloop*: I=∅. The call to *parseloop* (from *push*) is with GOTO$_k$`(I,nt)` as actual parameter. Like above, we can calculate this test statically in the special version of c-member. The function can be seen in figure 5.

```
c-member-nt(nt,nt-set,I,rej,c) =
 if (nt-set = ∅) 'rejected
 else
  let nt-stat:nts = nt-set
  in if ((nts = ∅) ∧ ¬rej) c(nt-stat)
  else
   let next-I = GOTO_k(I,nt-stat)
   in if ((next-I≠ ∅) ∧ (nt-stat=nt))
         c(nt-stat)
   else c-member-nt(nt,nts,I,
                    (next-I=∅) ∨ rej,
                    c)
```

Figure 5: *"The trick" for nonterminals*

The reader might argue that this improvement of the trick is a lot of work for a little improvement, but conditionals with all or almost all branches rejecting are very common, if the general version of "the trick" is used. Furthermore, if the sets $\mathcal{T}$ and $\mathcal{N}$ are large the above improvement decreases the size of the specialized parsers drastically.

7.3 Insertion of memoization points

Due to the polyvariancy and memoization of Similix, functions *shift* and *reduce* appear in many instances in the residual program — once for each set of static parameters. Some of the subexpressions of the different instances are α-equivalent. These duplicates can be eliminated by introducing *memoization points*. Memoization points introduce sharing in residual code of expressions used with the same values for its static free variables. Note that if a function generated this way turns out not to be shared, it is post-unfolded by Similix, so at worst the parser takes a little longer to generate. We use an operator MEMO for denoting insertion of a memoization point.

7.3.1 *Shift*

Function *shift* (figure 2) makes the transition from one state I_1 to another I_2 on input $t \in \mathcal{T}$. The whole transition is unique in the sense that we will only be in state I_1 having input t once in the specialized parser. But there might be

more than one transition to I_2, so if we insert a specialization point *after* the calculation of I_2, we can share some code. The new version of *shift* can be found in figure 6.

```
shift(I,κ^stack,t_1 ··· t_k:ts) =
  let κ^stack_new = push(I,κ^stack)
      I_new = GOTO_k(I,t)
  in MEMO parseloop(κ^stack_new,I_new,
                    t_2 ··· t_k:ts)
```

Figure 6: *shift as a specialization point*

7.3.2 *Reduce*

In *reduce* (figure 2), it is possible that two subexpressions appear in many instances. First the expression:

```
let κ^state=κ^stack(to-pop)
in κ^state(A,ts,κ^state)
```

can be shared since `to-pop` and A are the only free static variables. Hence, the expression can now be shared everywhere $A \rightarrow \alpha \cdot$ is reduced and even when $A \rightarrow \gamma \cdot$ is reduced if $|\gamma| = |\alpha|$.

In the `else`-branch the call to *parseloop* can be shared as in *shift*, though sharing will not happen very often in this case, since it only handles empty productions.

```
reduce((A→α·,β),I,κ^stack,ts) =
  let to-pop=|α|-1
  in if to-pop≥0
       MEMO let κ^state=κ^stack(to-pop)
            in κ^state(A,ts,κ^state)
     else let κ^stack_new=push(I,κ^stack)
              I_new = GOTO_k(I,A)
          in MEMO parseloop(κ^stack_new,I_new,ts)
```

Figure 7: *reduce as a specialization point*

8 Results

While the work described above was carried out using Similix 4.0 [4], we use Similix version 4.9, which is the current, experimental version of Similix, for the results of this section. We run Scheme programs in Chez Scheme 3.2, and use the timing facilities of Similix. All times exclude garbage collection. Programs are run on a SPARC station ELC. In these results we concentrate on $k = 1$, that is LR(1) parsers. We have chosen to use the following grammars:

$\mathcal{G}_1$, Full parenthesis — 4 productions.

$\mathcal{G}_2$, Expression with multiplication, division, addition, substraction and parenthesis — 8 productions.

$\mathcal{G}_3$, MIXWELL — the core language of the first selfapplicable partial evaluator Mix (described in [9]) — 19 productions.

8.1 Time needed to generate parsers

Generating *parsergenerator* using the equation *parsergenerator* = *cogen*(*g-parser*) takes 13.7s for the general LR(1) parser.

Figure 8 contains timings for generating specialized parsers using equation 1 of subsection 2.2:

$$parser_{\mathcal{G}} = mix(g\text{-}parser, \mathcal{G})$$

and using the *parsergenerator*:

$$parser_{\mathcal{G}} = parsergenerator(\mathcal{G})$$

	$mix(g\text{-}parser,\mathcal{G})$	$parsergenerator(\mathcal{G})$
$\mathcal{G}_1$	4.09s	1.82s
$\mathcal{G}_2$	26.4s	15.6s
$\mathcal{G}_3$	245s	201s

Figure 8: *Generating specialized parsers*

We see that using the parser generator speeds parser generation up between 1.22 and 2.25 times. Parser generation is slow — since efficiency had little priority when implementing the static primitive functions (GOTO, CLOSURE *etc.*), this is no great surprise.

8.2 Speedup gained by partial evaluation

In figure 9 we compare the runtimes of the general and specialized parsers on a selection of input. The third column contains runtimes for the original general parser presented in section 4. We have chosen to show these runtimes instead of the runtimes of the binding time improved parser; the general parser developed in section 7 will be slower due to the binding-time improvements, as these introduce extra (static) computations. We thus believe that our figures give a fair estimate of the speedups achieved by partial evaluation. The fourth column shows what is achieved, if the original (section 4) parser is specialized. The fifth column shows the runtimes for the specialized binding time improved parsers. The speedup in the last column is the ratio between the original parser and the specialized binding-time improved parser

	$\lvert\alpha\rvert$	†	‡	§	Speedup
$\mathcal{G}_1$	2	25.8ms	19.1ms	0.22ms	117
$\mathcal{G}_1$	8	80.1ms	70.1ms	0.48ms	167
$\mathcal{G}_1$	28	265ms	251ms	1.55ms	171
$\mathcal{G}_2$	3	104ms	77.0ms	0.44ms	236
$\mathcal{G}_2$	13	360ms	336ms	1.27ms	283
$\mathcal{G}_2$	35	921ms	851ms	2.82ms	327
$\mathcal{G}_3$	9	86.0ms	56.0ms	0.78ms	110
$\mathcal{G}_3$	51	936ms	913ms	2.43ms	385
$\mathcal{G}_3$	186	4.01s	3.99s	7.55ms	531
$\mathcal{G}_3$	983	20.0s	20.6s	31.5ms	635

†$Org\text{-}g\text{-}parser(\mathcal{G},\alpha)$
‡$Org\text{-}parser_{\mathcal{G}}(\alpha)$
§$parser_{\mathcal{G}}(\alpha)$

Figure 9: *Speedup by partial evaluation*

The figure shows that partial evaluation gives very large speedups, and that the speedups achieved is completely due to the binding-time improvements made. The large speedups stem mostly from the precalculation of the states (the GOTO function is calculated at specialization time). Constructing the states now can be done at specialization time, instead of being calculated (repeatedly) at runtime. Since the auxiliary functions of section 3 were not written with speed in mind, they are indeed time consuming, so the big speedup factors should not come as a great surprise.

Figure 10 shows the sizes of the specialized parsers. To avoid that sizes depend on the identifier names chosen, the figures are given as number of `cons`-cells as well as in Kbyte. We see that the parsers are relatively compact.

	$\mathcal{G}_1$	$\mathcal{G}_2$	$\mathcal{G}_3$
`cons`-cells	1561	3679	6199
Kbyte	7.87	18.7	30.2

Figure 10: *Size of specialized parsers*

8.3 Comparison with YACC

The generation of parsers using YACC is almost instant (approx. 0.1s, followed by the slower but still relatively fast C compiling) opposed to the very long parser generation time of our system. We expect that a considerable speedup of parser generation time can be achieved by clever optimization on the primitive functions, that are computed on parser generation time (= partial evaluation time). We have not emphasized this in our implementation. But we do not expect the parser generator to be able to compete with systems like YACC in this respect, since our method is inherently slower.

Figure 11 compares our specialized parsers with parsers generated by YACC.

	$\mid \alpha \mid$	LR(*1*)	YACC	$\frac{\text{LR}(1)}{\text{YACC}}$
$\mathcal{G}_2$	3	0.48ms	0.15ms	3.2
$\mathcal{G}_2$	13	1.09ms	0.26ms	4.2
$\mathcal{G}_2$	35	2.35ms	0.47ms	5.0
$\mathcal{G}_3$	9	0.62ms	0.20ms	3.1
$\mathcal{G}_3$	51	1.90ms	0.52ms	3.7
$\mathcal{G}_3$	186	6.20ms	1.47ms	4.2
$\mathcal{G}_3$	983	23.1ms	6.63ms	3.5

Figure 11: *Comparison with YACC*

The figures for the two systems cannot be measured in excactly the same way, but they are indeed comparable. Both figures excludes lexical analysis (in YACC this is achieved by letting every token be one character). To compare with YACC we run Chez Scheme at optimization level 3.

Still the figures indicate that our parsers are are only about 4 times as slow as parsers produced by YACC. It is promising, that our we get (slightly) better results for the larger grammer. The longer runtimes, we believe, is completely due to the slowness of Scheme compared to C.

9 Related Work

The only previous attempt (known to the author) of partial evaluation of general parsers is by Dybkjær [7]. A closely related problem is pattern matching; Consel and Danvy describe how the efficient Knuth-Morris-Pratt algorithm for string matching can be obtained by binding time improving and partially evaluating a naive matching algorithm [6]. Queinnec, Geffroy [15] and Jørgensen [10] describe more complex pattern matchers.

One of the first to describe the process of binding-time improvement was actually Dybkjær [7] describing partial evaluation of parsers. Standard binding-time improvements for Similix is described in the Similix Manual [4], and some more specific, aimed at partially evaluating an interpreter for a Miranda-like language, are described in [11]. In [13] the present author describes a debugging tool for doing binding-time improvements — the debugger can help finding the reason why variables are updated from static to dynamic. The debugger has been used heavily in this work. [8] contains a chapter describing binding-time improvements.

10 Conclusion and Future Work

We have developed a parser generator using partial evaluation. The parser generator is not fast, but the generated parsers are very efficient and have reasonable size. Parser generators have been obtained generating SLR (see [14]) and LR(1) parsers. In [14] we also discuss adding a failure continuation, in order to make the parser universal.

All this has been achieved by a number of transformations on the original program. These transformations have not altered the way the general parser works, only "moved" computations in order to get better binding-times.

The transformations used are applicable in other applications of partial evaluation. These can be summarized by:

- A static datastructure S under dynamic control, can be transformed into a function f_S, if the same function $f(s, X)$ is always applied to the elements s of S. f_S takes as first argument an index (in our case the number of elements to pop) and as second X.
- "The trick": if a dynamic variable v is known to belong to a statically computable finite set V, look up v in V, and continue with the static value.
- If many values in V yields the same value (in our case 'rejected), these cases can be collected in one `else`-branch.
- Insertion of specialization points, can give rise to more sharing in the residual programs.

It would therefore be interesting to examine to what extent these transformations might be automated.

A safe way to improve binding-times would be to (more or less) write the parser generator by hand, but we believe, that this kind of improvement has been avoided. The binding-time improvements have been done, knowing that the number of possible states is finite, and that it was thus a good idea, to make the current state static, in order to get kind of a DFA. Apart from this we believe, that we have not used any knowledge of table driven parsers.

The process of binding-time improving was however rather tedious, and a library of transformations (either as

descriptions or preferably as mechanized aids) is crucial, before great advantages are gained by using partial evaluation instead of doing all the programming by hand.

The amount of programming done is not much smaller, than it would have been, had we written the parser generator by hand. The programming has however been eased by the fact, that we did not have to think of "parsergeneration time" and "parse time". This also makes it much easier to experiment with the parser: the transformations of section 5 and 6 improving the performance of generated parsers are easier to do on the general parser than on a parser generator. We believe (without evidence), that the specialized parsers generated by our system is faster than specialized parsers generated by a naive hand-written LR parser generator.

We conclude that it has been worthwhile to use partial evaluation for generating parser generators: partial evaluation makes it easier to consider the essential parts of the problem, and as a consequance many optimizations, which would seem difficult to do in a parsergenerator, are straightforward.

Acknowledgements

I would like to thank Torben Mogensen and Anders Bondorf for many fruitful discussions. Thanks also goes to Neil Jones for inspirering me to take up this project.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*. IFIP TC2, North-Holland, 1988. Workshop proceedings, October 1987, Gl. Avernæs, Denmark.

[3] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming,*, 17(1-3):3–34, Dec. 1991. Revision of paper in ESOP'90, LNCS 432, May 1990.

[4] A. Bondorf. *Similix Manual, system version 4.0*. DIKU, University of Copenhagen, Denmark, Sept. 1991. Included in Similix distribution.

[5] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[6] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark. Lecture Notes in Computer Science 432*, pages 88–105. Springer-Verlag, May 1990.

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

[8] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[9] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202*, pages 124–140. Springer-Verlag, 1985.

[10] J. Jørgensen. Generating a pattern matching compiler by partial evaluation. In P. C. van Rijsbergen, editor, *Glasgow Workshop on Functional Programming, Ullapool*, pages 177–195, Glasgow University, July 1990. Springer-Verlag.

[11] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 258–268, Jan. 1992.

[12] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607 – 639, 1965.

[13] C. Mossin. Similix binding time debugger manual, system version 4.0. Included in Similix distribution, Sept. 1991.

[14] C. Mossin. Partial evaluation of general parsers. Student Report 92–8–1, DIKU, University of Copenhagen, Denmark, Aug. 1992.

[15] C. Queinnec and J.-M. Geffroy. Partial evaluation applied to pattern matching with intelligent backtrack. In *Workshop on Static Analysis*, pages 109 – 117, 1992.

[16] D. A. Schmidt. *Denotational Semantics, a Methodology for Language Development*. Allyn and Bacon, Boston, 1986. 1988: Wm. C. Brown Publishers.