A Self-Applicable Partial Evaluator for Term Rewriting Systems

Anders Bondorf
DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark (*)
uucp: anders@diku.dk

Abstract

This paper describes a fully self-applicable partial evaluator developed for equational programs in the form of term rewriting systems. Being self-applicable, the partial evaluator is able to generate efficient compilers from interpreters as well as a compiler generator automatically.

Earlier work in partial evaluation of term rewriting systems has not achieved self-applicability due to the problem of partially evaluating pattern matching. This problem is overcome by developing an intermediate language for being able to express pattern matching at an appropriate level of abstraction.

We describe the intermediate language and partial evaluation of it. Binding time analysis, a well-known preprocessing technique, is used. We introduce further preprocessing to deal efficiently with our intermediate language.

The system has been implemented and compilers for small languages as well as a compiler generator have been generated with satisfactory results.

Keywords

Decision trees, functional languages, pattern matching, elementary matching operations, binding time analysis, abstract interpretation, partially static structures.

(*) Until July 1989:
University of Dortmund, Lehrstuhl Informatik V
Postfach 50 05 00, D-4600 Dortmund 50
Federal Republic of Germany
uucp: anders@unidoi5.ls5.informatik.uni-dortmund.de

1. Introduction

The potential use of partial evaluation for doing compilation, compiler generation, and even compiler generator generation has been known since the early seventies [Futamura 71]. A few years ago these promising ideas were carried out for the first time in practice in the LISP based "Mix" project in the Copenhagen group around Neil D. Jones [Jones, Sestoft, & Søndergaard 85]. Various people have since then been working on partial evaluation in Copenhagen and other places. The aims have been to understand partial evaluation better and to develop stronger partial evaluators: to make them "as automatic as possible" and to use stronger languages.

Partial evaluation is a general program transformation which, given a *subject* program and *static* values of some but not all of its input parameters, produces a so-called *residual* program [Ershov 82]. This, when applied to the rest of the inputs, will yield the same result the original program would have yielded on all its inputs. Partial evaluation is thus *program specialization*: its effect is to yield a new program equivalent to the original on a certain subset of its input. We therefore also refer to a partial evaluator as a *specializer*.

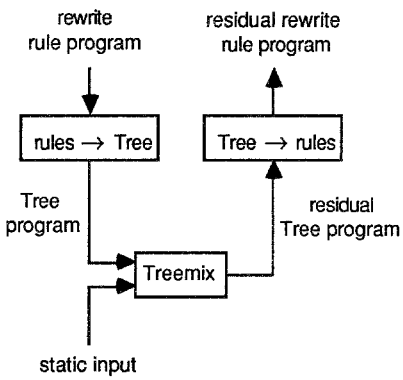
To compile by partial evaluation, an interpretive specification of the language is needed. By specializing the interpreter with static input being the interpreted source program, a target program is produced (compilation). And by *self-application*, specialization of the specializer itself with static input being an interpreter, a stand-alone compiler is generated. Finally, by specializing the specializer with the static input being the specializer itself, a compiler generator is produced.

Equational programming, that is programming with *term rewriting systems* [Dershowitz 85] [Huet & Levy 79] [Huet & Oppen 80], provides a convenient formalism for defining computations. In particular, as described in [Hoffmann & O'Donnell 82] and also [Turchin 86] (for the language REFAL), equational programming can be used for defining interpretive language specifications. It has therefore been a natural goal to realize the ideas of partial evaluation in the context of term rewriting systems: to use self-application to transform interpreters written in equational style to compilers, also written in equational style. This transformation has been performed for other languages (first time: [Jones, Sestoft, & Søndergaard 85]), but to our knowledge never before for term rewriting systems.

Self-application means that the specializer plays two roles: as specializer and as subject program. The specializer therefore has to be written in the same language as the language of the programs it treats. Such a specializer is called an *autoprojector* [Ershov 82]. Experience from the "Mix" project has shown that successful specialization of a *self-interpreter* (an interpreter written in the same language as it interprets) is a first step towards self-application. This is indeed plausible as one may consider a specializer as being a "smart" interpreter: to evaluate static expressions, it contains the code of an interpreter. An autoprojector is a "self-specializer", and thus a "smart" self-interpreter. Therefore, if an autoprojector performs badly when specializing a self-interpreter, then it cannot be expected that it will ever be able to specialize itself (self-application) in a satisfactory way.

We have already described partial evaluation of a subclass of term rewriting systems in [Bondorf 88]. The partial evaluation methods described there gave good results in a number of cases, including specialization of a non-trivial interpreter (for a small lambda calculus based language with higher order functions). However, when a self-interpreter for the term rewriting system language was specialized, the partial evaluator did not perform well: enormous specialized (self-) interpreters resulted due the way of dealing with *pattern matching*. Since a self-interpreter could not be specialized satisfactorily, self-application was out of question.

In this paper we describe an approach which has achieved self-application: rather than directly specializing a program in the form of a term rewriting system, we first translate the program into an *intermediate form*. This program is then specialized yielding a residual program, *also* in intermediate form. Finally, the residual program is translated back into a term rewriting system:



(*Tree* is the name of the intermediate language; the specializer is called *Treemix*.) In the intermediate language, pattern matching is expressed in the form of so-called *decision trees* (or *matching trees* [Huet & Levy 79]): pattern matching has been factorized into explicit primitive operations for comparing values and decomposing data structures. But the intermediate language still contains enough structure to make it possible to perform the translation back into pure rewrite rule form. LISP or, say, assembler language (!) as intermediate language would not suffice here. One could of course in principle translate a residual program written in e.g. LISP into rewrite rule form, but the result would hardly be readable. Consequently, existing LISP based partial evaluators [Jones, Sestoft, & Søndergaard 88] [Mogensen 88] cannot be used for our purpose.

Our concrete decision tree language basically is a *functional* language, but its control primitives differ from those of for instance Mixwell [Jones, Sestoft, & Søndergaard 88]. Furthermore, to match the intension of being an intermediate language for term rewriting systems, there are certain restrictions imposed on the allowed expression forms. To partially evaluate decision tree programs, well-known techniques for partial evalua-

tion of functional programs [Jones, Sestoft, & Søndergaard 88] can be used as a start point, but they must be modified to deal with the new control primitives and restrictions.

Of the two hardest non-trivial problems of partial evaluation, those of *termination* and *self-application*, the main concern of this paper is self-application. We therefore address *binding time analysis*, and more generally *preprocessing*, in some detail. Preprocessing is essential for efficient self-application [Bondorf, Jones, Mogensen, & Sestoft 89].

1.1 Outline

In section 2 we summarize definitions and terminology. In section 3 we argue for using decision trees; our concrete decomposition tree language is described in section 4. Section 5 describes partial evaluation of decision tree programs, and it ends with a brief discussion of finiteness. Section 6 is devoted to preprocessing. Section 7 contains an overview of the results. In section 8 we mention related work, and in section 9 we conclude.

2. Definitions and terminology

In this section we review some basic concepts of partial evaluation and term rewriting systems.

2.1 Programming languages and program specialization

Let V be a universal domain containing programs and data values, including a specific element [], and suppose v1 : v2 is in V for any v1, v2 in V. An example is the set of all Lisp S-expressions. As in ML we write [x1, x2, ..., xn] to stand for x1 : [x2, ..., xn] so [x1, x2, ..., xn] = x1 : x2 : ... : xn : [], where : associates to the right.

A *programming language*, for instance L_i, is a partial function that maps programs to meanings, which themselves are partial functions from input to output:

$$L_i: V \to (V \to V)$$

An L_i-*program* p is any $p \in V$ such that $L_i\ p$ is defined. Each program will take *one* input which, however, may be a list.

Suppose p is an L_2-program expecting inputs of the form [vs, vd]. A *residual program for* p *with respect to* vs is an L_3-program p_{vs} satisfying

$$L_3\ p_{vs}\ vd = L_2\ p\ [vs, vd]$$

We also say that p_{vs} is p *specialized with respect to* vs, and vs is called the *static* (often referred to as "known" or "available") input, while vd is the *dynamic* ("unknown") input.

A *partial evaluator* (or *program specializer*) is an L_1-program mix such that L_1 mix [p, vs] is a residual program for every p, vs. This can be re-expressed by the "mix equation":

$$L_3\ (L_1\ mix\ [p, vs])\ vd = L_2\ p\ [vs, vd]$$

p is called the *subject* program of partial evaluation. For mix to be an autoprojector, $L_1 = L_2$. We shall only be interested in the case where also $L_2 = L_3$, so $L_1 = L_2 = L_3 = L$. In our case L is the decision tree language.

2.2 Interpreters and compilers

Let $S: V \to (V \to V)$ be a programming language, perhaps different from L. Given an S-program source that maps some input data to an output result, an *interpreter for* S *written in* L is an L-program int such that for all source and data

$$L\ int\ [source, data] = S\ source\ data = result$$

An S-*to*-L *compiler written in* L is an L-program comp mapping source programs written in S onto target programs written in L:

$$target = L\ comp\ source$$

where target must satisfy

$$L\ target\ data = S\ source\ data = result$$

2.3 Compilation and compiler generation

The program specializer mix described above may be used to compile from S to L by specializing int with respect to source:

$$target = L\ mix\ [int, source] = int_{source}$$

By the mix equation it is easy to see that this target program satisfies the requirement:

$$L\ target\ data = S\ source\ data = result$$

If L = S, then int is a *self-interpreter* sint. In this case

$$L\ sint_{source}\ data = L\ source\ data$$

for all data.

By self-applying the partial evaluator, we can generate a compiler from the interpreter:

$$comp = L\ mix\ [mix, int] = mix_{int}$$

When given a source program, this comp produces a target program (this follows immediately from the mix equation):

$$L\ comp\ source = L\ mix\ [int, source] = target$$

We define that

$$cogen = L\ mix\ [mix, mix] = mix_{mix}$$

and see that when given an interpreter, cogen produces the compiler: comp = L cogen int.

2.4 Term rewriting systems

We assume some knowledge about term rewriting systems, but we shall here shortly review some concepts in order to set up a terminology.

We consider a set of *variables* V, and a set of *operators* Σ such that for all op$\in\Sigma$: arity(op)$\geq$0 (an arity is also called a *rank*). The set of *terms* $T_\Sigma(V)$ generated by Σ over V is defined such that: 1) every variable is a term; 2) every operator op$\in\Sigma$ for which arity(op)=0 is a term; 3) if $t_1, ..., t_n$ are terms and if there exists an operator op$\in\Sigma$ for which arity(op)=n, then op($t_1, ..., t_n$) is a term. The set of *ground terms* T_Σ is defined as the set of terms without variables. Terms with variables are called *open* terms.

A *term rewriting system* is a set *rewrite rules*, which may be *ordered* (in which case the system is a *priority rewrite system* [Baeten, Bergstra, & Klop 87]). A rewrite rule is a pair of terms (l_i, r_i) with the restriction that l_i is not a variable; it is written as $l_i \rightarrow r_i$ where l_i is called the *left-hand side* and r_i is called the *right-hand side*. A further restriction is that no variable must occur in any r_i unless it also occurs in the corresponding l_i. A term is *linear* if no variable occurs more than once in it. A (*left-*) *linear rewrite rule* is a rewrite rule in which the left-hand side is linear.

Sometimes Σ will be divided into two disjoint sets, $\Sigma = \Delta + \Gamma$. Δ is the set of *function symbols* (or *defined operators*) and Γ is the set of *constructors*. The set T_Γ is denoted as the set of *ground constructor terms*. A *functional term rewriting system* is a term rewriting system in which the left-hand sides are restricted to be of the form $\delta(p_1, ..., p_n)$ ($n\geq0$) where $\delta\in\Delta$ and all $p_i\in T_\Gamma(V)$, that is, they do not contain function symbols. Functional term rewriting systems correspond to "systems with constructors" [Huet & Levy 79].

3. Motivation for decision trees

Successful self-application can only be expected if a self-interpreter can be specialized satisfactorily. In this section we motivate the use of decision trees rather than term rewriting systems by arguing that specialization of a rewrite system self-interpreter yields large residual programs, whereas good results are achieved for decision trees.

Recall (section 2.3) that for a self-interpreter sint and for an arbitrary L-program source

$$L\ sint_{source}\ data = L\ source\ data$$

for all data. $sint_{source}$ and source are thus programs which compute the same function, and they are also written in the same language, L. We may therefore expect that they also *textually* are "almost equal": $sint_{source} \cong source$. In other words, if $sint_{source}$ is significantly bigger or runs significantly slower than source, then the self-interpreter was not specialized satisfactorily. In that case we cannot expect successful self-application of the specializer.

3.1 Specialization of a self-interpreter for term rewriting systems

A self-interpreter for rewrite rule programs contains some code for matching a term against a pattern (a left-hand side of a rule). This code traverses the term and the pattern in parallel and yields either a substitution for the variables in the pattern (in the case of a successful match) or a mismatch. If a mismatch occurs, then the matching algorithm is reapplied to the term and another pattern. The process continues until a successful match is found (if it exists). The algorithm is of course inefficient; in addition to this, specialization of it gives undesired results, as we shall now see.

Let us consider a small piece of a (functional) term rewriting system source:

```
f(a, b)  → one                    rule f1
f(a, c)  → two                    rule f2
...
...      → ... f(X, Y) ...
```

Here f is a function symbol, a, b, and c are constructors, and X and Y are variables.

Specialization is performed as a *symbolic evaluation* over a domain of terms containing variables representing the dynamic unavailable values. During specialization of the self-interpreter with respect to source, the self-interpreter is going to handle the term f(X, Y). The self-interpreter calls its matching algorithm to match f(X, Y) against the two f rules. But now we are *specializing* the self-interpreter, not executing it, so we (the specializer) do not know the values bound to X and Y (they depend on the unavailable input data); we only know the patterns from the two f rules in source. The specializer therefore produces a *specialized version* of the matching algorithm: it can match a term against the two f rules. The specialized matching algorithm will be something like this:

```
match(f1)(X, Y) →
  if X = a then
    if Y = b then one             successful match
    else match(f2)(X, Y)          try next rule
  else match(f2)(X, Y)            try next rule

match(f2)(X, Y) →
  if X = a then
    if Y = c then two             successful match
    else ...                      failing match
  else ...                        failing match
```

The naive if-then-else structure has been inherited from the "trial and error" matching algorithm of the self-interpreter. In pure rewrite rule form, with all cases listed, we get the following priority rewrite system (ordered from top to bottom):

```
match(f1)(a, b) → one             successful match
match(f1)(a, Y) → match(f2)(a, Y) try next rule (i)
match(f1)(X, Y) → match(f2)(X, Y) try next rule (ii)
match(f2)(a, c) → two             successful match
match(f2)(a, Y) → ...             failing match
match(f2)(X, Y) → ...             failing match
```

By *unfolding* the $match_{(f2)}$ terms in *(i)* and *(ii)* through *instantiation* (backwards substitution) of the variables X and Y for all relevant cases, we get:

```
match(f1)(a, b) → one             successful match
match(f1)(a, c) → two             successful match (i)
match(f1)(a, Y) → ...             failing match (i)
match(f1)(a, Y) → ...             failing match (i)
match(f1)(a, c) → two             successful match (ii)
match(f1)(a, Y) → ...             failing match (ii)
match(f1)(X, Y) → ...             failing match (ii)
```

Such instantiations correspond to Turchin's *contractions* [Turchin 86]. We note that overlapping left-hand sides make instantiation followed by unfolding semantically problematic.

Now $match_{(f1)}$ (in $sint_{source}$) closely corresponds to f (in source), but redundant rules have been generated due to the structure of the self-interpreter. The number of such extra rules depends on the *product* of the number of $match_{(fx)}$ rules for every left-hand side x of f in source! This is of course completely unacceptable for realistic programs; we do *not* achieve $sint_{source} \cong$ source. The above program may be reduced by removing redundant rules; however, to detect these a rather complex machinery is needed. Experience has shown that unacceptably slow partial evaluation results.

3.2 Specialization of a self-interpreter for decision trees

Let us now consider a decision tree for the program piece in source (we omit a formal definition of decision trees; the semantics is the obvious one):

```
f(X, Y) →
  case X of
    a: case Y of
         b: one                   successful match
         c: two                   successful match
       end
  end
```

A self-interpreter for a decision tree language does not have to deal with mismatch cases. The decision tree guides the pattern matching: it is now a simple parallel search in the decision tree and the term. If a mismatch is found, then the term does not match *any* rule. We choose to consider this as an error, a *match error*, so it is semantically correct for the self-interpreter not to care about mismatches at all. A match error at the level of the interpreted program source will be reflected as an error at the level of the self-interpreter itself.

The specialized matching algorithm now looks like this:

```
match(f)(X, Y) →
  if X = a then
    if Y = b then one          successful match
    else
      if Y = c then two        successful match
      { no "else" }
  { no "else" }
```

If no match is found, the absence of "else" branches results in an error. By converting the equality tests into case dispatches, we get:

```
match(f)(X, Y) →
  case X of
    a: case Y of
         b: one                successful match
         c: two                successful match
       end
  end
```

This *exactly* corresponds to the original piece of source. This should make it plausible that it is possible to achieve $sint_{source} \equiv$ source for decision trees.

4. The decision tree language Tree

As indicated by the previous example, the idea is to translate a set of left-hand sides into a decision tree (with the right-hand sides at the leaves) in which pattern matching is factorized into a series of *elementary matching operations*. The choice of pattern matching primitives is a compromise between two contrasting requirements: strong primitives close to the rewrite rule form are desirable for the translations to and from rewrite rule form. But simple primitives are desirable for partial evaluation.

4.1 Syntax and semantics

Basically, the language Tree is a *statically scoped, untyped* and *first order* functional language that uses *innermost* (call-by-value, strict) *deterministic* reduction. Other reduction strategies like normal order reduction (lazy evaluation) could also be defined, but this would require a complete revision of the partial evaluator; partial evaluation depends strongly on the operational properties of a language. Innermost reduction is the simplest to deal with for at least two reasons: 1) there are no infinite data structures; 2) pattern matching operations do not influence redex reduction as in lazy pattern pattern matching.

The data structures are Lisp S-expressions built from an in principle infinite set of *0-ary constructors* (like Lisp atoms and Prolog 0-ary functors) and one fixed *binary constructor* "." (like Lisp "cons" or Prolog "."). Arbitrary constructors could be introduced as syntactic sugar (together with some kind of type checking system), but they have not been included in the core language. Allowing arbitrary constructors would necessitate the *encoding* of programs when they are used as input to a self-interpreter or to the partial evaluator. Actually, when specializing a self-interpreter or the specializer itself with respect to a program, it is necessary to encode this program *twice*. By disallowing arbitrary constructors, we avoid all encoding and decoding problems. Notation: as in LISP, we use $(a_1 \ldots a_n)$ as shorthand for $(a_1 . (\ldots . a_n))$, and 'a as shorthand for (quote a).

The (abstract) syntax of Tree follows:

Syntax

P ∈	`Program`	*programs*
F ∈	`Function`	*functions*
B ∈	`Body`	*bodies*
A ∈	`Alternative`	*alternatives*
P ∈	`Pattern`	*patterns*
R ∈	`R`	*right-hand sides*
V ∈	`Var`	*variables*
N ∈	`Name`	*function names*
C ∈	`Cst`	*constants*
S ∈	`Symbol`	*0-ary atomic symbols*

P ::= (F^+)
F ::= (N (V^*) B)
B ::= (case V A^*) | (equal V_1 V_2 B_1 B_2) | R
A ::= (P → B)
P ::= 'C | (V_1 . V_2) | else
R ::= 'C | (R_1 . R_2) | V | (call N R^*) | (xcall N R^*)
C ::= S | (C_1 . C_2)

One function corresponds to one decision tree.

Pattern matching is performed by case and equal. case is used for matching a term against a constructor, equal for comparing two terms for

equality. The case alternatives are matched in a strictly *deterministic order* from top to bottom until a match is found. The body corresponding to the matched pattern is then evaluated. If no pattern matches, an error occurs and the evaluation stops with a *match error*. If more than one pattern matches (corresponds to overlapping left-hand sides in a term rewriting system), only the first one is considered (thus imposing a priority). A term matches a pattern 'C if the term is equal to the constant C. A term matches a pattern $(V_1 . V_2)$ if it is a pair, i.e. a term constructed with the binary constructor. The left part (the "car") of the term is then bound to V_1, the right part (the "cdr") to V_2. An else pattern is an "always match". The equal construction is a simple kind of conditional (useful for dealing with non-linear rewrite rules). Read it as "if $V_1 = V_2$ then B_1 else B_2".

The case construction is similar to "case expressions *without* default/fail clauses" [Augustsson 85] [Peyton Jones 87]. These have a particularly simple flow of control, which is desirable for partial evaluation. The drawback is that duplication of right-hand sides sometimes occurs when a rewrite rule program is translated into decision tree form.

The (non-nested) R parts correspond to right-hand sides of rewrite rules; they constitute the *leaves* of the decision tree. Points to notice are that quote is used in the usual Lisp way to denote constants (ground constructor terms), that function symbols are preceded by the keyword call (so redexes, or *calls*, are identified syntactically), that external functions are available through xcall, and finally that the pairing operator is written as ".". Functions have fixed arity; type correctness with respect to this is checked statically.

The distinction between the syntactic forms Body and R implies that all pattern matching tests must occur in the *beginning* of a function body. This reflects the intension with the language as being an intermediate form for rewrite rule programs. In these, the operations of matching and evaluation of the right-hand side of the matched rule are completely separated. Also, *only variables*, not arbitrary expressions, may be tested by case and equal. We thus disallow the possibility of specifying evaluation of an arbitrary expression during pattern matching. This reflects that pattern matching is a "passive" process, a search for a match giving a substitution. It does not itself compute. (Note: lazy pattern matching is also "passive" in this sense, even though it may force evaluation of a suspended call in the term.)

4.2 Example

For an example, let us consider a program piece that tests whether a given (ground constructor) term t matches a given pattern p. Let the terms be S-expressions and let the patterns be either constants, pairs, or variables:

Pat = (C . Const) | (Pat . Pat) | Variable

A variable is an atomic symbol different from C. The program piece is the following one:

```
(match (p t)
  (case p
    ((p1 . p2) →
      (case p1
        ('C → (equal p2 t 'true 'false))          constant
        (else →                                   pair
          (case t
            ((t1 . t2) →
              (call and (call match p1 t1) (call match p2 t2)))
            (else → 'false)))))
    (else → 'true)))                              variable

(and (b1 b2) (case b1 ('false → 'false) (else → b2)))
```

The deterministic pattern matching distinguishes constant from pair patterns, and equal compares a term and a constant pattern.

4.3 Translation to and from Tree

The problem of developing and discussing efficient algorithms for translating from rewrite rules to decision tree form and vice versa is outside the scope of this paper. However, we must ensure that Tree is strong enough to be useful as an intermediate language. The most severe restriction of Tree is its fixed innermost reduction strategy; we only address the problem of representing term rewriting systems with innermost reduction. We cannot expect Tree to be suitable for handling normal order reduction (nor other reduction strategies).

We first observe that it is possible to translate any *functional* term rewriting system based on S-expression data types (possibly with non-linear as well as prioritized rules) into Tree: group together all rewrite rules with left-hand sides rooted by the

same function symbol, while keeping the specified priority. Then translate each group into a Tree function. This is always *possible* as one may generate the naive (and indeed inefficient) program, which just does naive pattern matching, testing one rule at a time. For certain restricted classes of term rewriting systems, there exist much better algorithms [Augustsson 85] [Peyton Jones 87].

The other way around, we see that any Tree program can be translated into a (functional) rewrite system: for each decision tree (function), generate one rewrite rule for each leaf. The order of the rewrite rules must be the same as the order of the leaves (when considering the "flattened" decision tree). For instance for the match program, this translation gives the following non-linear priority rewrite system (written in a syntax close to Tree):

```
(match ('C . p2) p2) → 'true                          constant
(match ('C . p2) t)  → 'false                         constant
(match (p1 . p2) (t1 . t2)) →                         pair
   (call and (call match p1 t1) (call match p2 t2))
(match (p1 . p2) t) → 'false                          pair
(match p t) → 'true                                   variable
(and 'false b2) → 'false
(and b1 b2)  → b2
```

When decision trees contain duplicated leaves, sophisticated algorithms may generate rewrite systems with fewer rules than leaves.

General (non-functional) term rewriting systems can be mapped into functional ones, and therefore also into Tree. The idea is to split operator occurrences into constructor and function symbol occurrences by performing the following steps: (1) replace all nested occurrences of left-hand side operators by constructors; (2) replace all right-hand operators by function symbols; (3) for all function symbols now occurring in any right-hand side, add a rule, with lowest priority, that replaces the function symbol by a constructor. For instance, let us consider a rewrite system for combinator logic (the example comes from [Klop 87]):

```
@(@(@(S, f), g), x) → @(@(f, x), @(g, x))
@(@(K, x), y)       → x
@(I, x)             → x
```

Here @ is an operator for *function application*; S, K, and I are operators, f, g, x, and y are variables.

By splitting the operator occurrences, we get a functional priority rewrite system (in Tree-near syntax):

```
(@ ('@ ('@ 'S f) g) x) → (call @ (call @ f x) (call @ g x))
(@ ('@ 'K x) y) → x
(@ 'I x)        → x
(@ x y)         → ('@ x y)
```

The functional version of the rewrite system operates on data structures with constructors rather than operators, but otherwise the behavior is the same. The systematic replacement of nested left-hand side operators by constructors is correct since, for *innermost* reduction, data structures only contain constructors, not function symbols. The extra rules replace function symbols by constructors; they are applied to terms (calls), which do not match any of the original rules (the extra rules have the lowest priority). Such calls are thus replaced by data structures; this "records" that the call is in normal form.

5. Partial evaluation of Tree

Partial evaluation or, more specifically, *polyvariant program specialization* [Bulyonkov 88] can be viewed as *abstract interpretation* over *open* terms with variables representing the unavailable dynamic data. For each Tree function, the abstract interpretation gives a set of possible variants. A *variant* associates with each function parameter an open term; a variant represents an *instance* of the function, *specialized* according to the constant static parts in the open terms. A residual program thus is a set of specialized functions. In a specialized function, operations depending only on static values have been reduced. In particular, if the test of a conditional can be decided, only the selected branch is present.

The residual program can be optimized by *unfolding* calls to the (specialized) functions. Abstract interpretation over open terms and unfolding is usually intermingled. Function calls are thus also unfolded "on the fly" during abstract interpretation, and we refer to the process as *symbolic evaluation*. Our specializer Treemix processes its subject program by performing such an evaluation. [Sestoft 86] describes symbolic evaluation in detail; a comprehensive discussion of polyvariant program specialization is found in [Jones 88].

In Tree there are certain restrictions on the allowed forms of the control expressions case and equal: only variables may be tested, and all tests must occur in the beginning of a function body. These restrictions do not have equivalents in e.g. LISP. The first restriction implies that some care must be taken when processing case and equal. Both restrictions have consequences for call unfolding. We address these problems in some detail.

5.1 Processing case and equal

We shall here give a piece of the algorithm for symbolic evaluation of Tree expressions. Given an expression of the form `Body`, the algorithm produces a residual body in which static tests have been performed. The algorithm is given in a style near to denotational semantics. It operates over the syntactic domains given earlier (`Body`, `R`, ...); we use Body, R, ... to denote the residual equivalents. Some notation: this typeface is used for pieces of the subject program being specialized and for the pieces of the residual program being generated.

A piece of the symbolic evaluation algorithm

Semantic domains

Residual expressions:

b:	Body	*bodies*
	R	*right-hand sides*
v, w:	Var	*variables*
c, d:	Cst	*constants*

Other domains:

t, u:	Open = Var + Cst + Pair	*open terms*
	Pair = Open × Open	*pairs*
θ:	Θ = (Var × Open)*	*unifiers*
ψ:	Ψ = (Var × (Var + Var × Var + Cst))*	*factorized unifiers*
	Fail = Unit	*failing unification*
th:	Θ + Fail	*unifier or fail*
e:	Env = Var → Open	*environments*

Symbolic evaluation functions

B: `Body` → Env → Body

B⟦(case V (P1 → B1) ... (Pn → Bn))⟧e =
  cases e(⟦V⟧) of
    isVar(v) → *generate test*
      (case v (P1 → B⟦B1⟧e) ... (Pn → B⟦Bn⟧e))
    isCst(c) → *perform test*
      if ∃i: (Pi = 'C such that c = C⟦C⟧) ∨
             (Pi = else) then
        let j be the smallest such i in B⟦Bj⟧e
      else (case) *always produces a match error*
    isPair(t, u) →
      A⟦(P1 → B1) ... (Pn → Bn)⟧ (t, u) e

B⟦(equal V1 V2 B1 B2)⟧e =
  cases e(⟦V1⟧), e(⟦V2⟧) of
    isCst(c), isCst(d) →
      if c = d then B⟦B1⟧e else B⟦B2⟧e
    isCst(c), isVar(v) or isVar(v), isCst(c) →
      (case v ('c → B⟦B1⟧e) (else → B⟦B2⟧e))
    isVar(v), isVar(w) →
      (equal v w B⟦B1⟧e B⟦B2⟧e)
    otherwise → *compound test*
      let th = U(e(⟦x⟧), e(⟦y⟧)) in
        cases th of
          isFail() → B⟦B2⟧e
          isΘ(θ) → G(T(θ), B⟦B1⟧e, B⟦B2⟧e)
        end

B⟦R⟧e = R⟦R⟧e

A: `Alternative`* → Pair → Env → Body

A⟦('S → B) A*⟧ (t, u) e = A⟦A*⟧ (t, u) e

A⟦('(C1 . C2) → B) A*⟧ (t, u) e =
  let th = U((C⟦C1⟧, C⟦C2⟧), (t, u)) in
    cases th of
      isFail() → A⟦A*⟧ (t, u) e
      isΘ(θ) → G(T(θ), B⟦B⟧e, A⟦A*⟧ (t, u) e)
    end

A⟦((V1 . V2) → B) A*⟧ (t, u) e =
  B⟦B⟧ [⟦V1⟧↦ t][⟦V2⟧↦ u]e

A⟦(else → B) A*⟧ (t, u) e = B⟦B⟧e

A⟦ ⟧ (t, u) e = (case)

R: `R` → Env → R *(definition omitted)*

C: `Cst` → Cst *(definition omitted)*

U: Open × Open → Θ + Fail
unifies two open terms (definition omitted)

T: Θ → Ψ

factorizes a unifier (definition omitted)

G: $\Psi \times Body \times Body \rightarrow Body$
generates a residual body (definition omitted)

The algorithm decomposes tests on compound open terms to ensure that only variables are tested in the residual body. This decomposition is performed by the functions U, T, and G. We shall here explain the decomposition along with an example.

Let us consider two open terms, t_0 and u_0, respectively a variable and a pair of a variable and a constant: $t_0 = p$, $u_0 = (q\ .\ '6)$. The function U unifies two terms. It produces a list of variable-value pairs rather than a function as we are interested in its actual bindings, not its extensional behavior. Identity bindings are not represented. For the example, $U(t_0, u_0) = \theta_0 = [(p, (q\ .\ '6)]$. Every variable-value pair represents an equality which must hold for two unified terms t and u to be equal. Such an equality corresponds to a test on a variable. The function T factorizes the unifier by introducing fresh variables for components of pair values. For example, $T(\theta_0) = \psi_0 = [(p, (r\ .\ s)), (q, r), (s, '6)]$. Now every test can be expressed in Tree, either by case or equal. For the example we get $G(\psi_0, b_1, b_2) =$

```
(case p
  ((r . s) →
    (equal q r
      (case s ('6 → b1) (else → b2))
      b2))
  (else → b2))
```

for some arbitrary bodies b_1 and b_2. We observe that since Tree has no "fail clauses" [Peyton Jones 87], code duplication of b_2 occurs. This has not caused practical problems in our experiments.

5.2 Call unfolding

Call unfolding is the process of replacing a function call by the body of the function, with the formal parameters replaced by the argument expressions in the call. The restrictions in Tree have some implications for call unfolding: (1) that only variables may be tested imply that some function calls must not be unfolded; (2) that all tests must occur in the beginning of a function body sometimes necessitates reorganizing expressions after call unfolding.

Problem (1) occurs if an actual parameter to a function call is itself a call. Unfolding the outer call may cause the inner call to be "caught" in a case or an equal test. For instance, let us consider the program piece ... (call f (call g y)) where f is defined by (f (x) (case x ...)). Unfolding the call yields ... (case (call g y) ...), which is a disallowed form as only variables may be tested. We thus cannot unfold the f call. This problem reflects a property of pure rewrite programming: whenever a computed value needs to be tested, a surrounding call to an auxiliary function handling the result is needed.

Problem (2) occurs if a non-tail call (i.e. a call being an argument to the pairing operator) is unfolded to an expression with tests. If, for instance, f is defined by

```
(f (x) (case x ((a . b) → (case a ('1 → 'one)))))
```

we might unfold the call in a pairing expression ... (y . (call f z)) to get

```
... (y . (case z ((a . b) → (case a ('1 → 'one)))))
```

This expression does not obey the restriction that all tests must occur in the beginning of the function body, but the unfolding obviously is semantically correct if one defines the unfolded expression in an extended language without the restriction. An expression like the above one can, however, always be converted into a semantically equivalent and syntactically allowed one. This is done by changing the order of the pairing and testing operations. For the example this yields:

```
... (case z ((a . b) → (case a ('1 → (y . 'one)))))
```

which is an allowed syntactic form. Such transformations are semantically correct due to our strict evaluation order. With a lazy semantics, the transformed version above would be less terminating than the non-transformed one.

5.3 Finiteness

Polyvarint program specialization gives two kinds of termination problems: (1) generation of infinitely many function variants, also known as *infinite specialization*; (2) infinite call unfolding. Infinitely many variants may be generated since the set of static values is infinite. Infinite unfolding may occur if the unfolding strategy is too liberal; infinite unfold-

ing may thus happen even if there are only finitely many variants.

Infinite specialization can be solved by *generalization* [Turchin 88] of static values: replace static values by variables during symbolic evaluation. Infinite call unfolding is avoided by choosing an unfolding strategy, which is "conservative enough". Since some call unfolding usually is performed during symbolic evaluation, infinite specialization may show up at first as infinite unfolding. Choosing a more conservative unfolding strategy in that case just results in the generation of infinitely many residual functions.

Detecting whether infinite specialization may occur is related to the halting problem and is in general undecidable. Jones addresses the problem in detail and develops algorithms, which ensure termination [Jones 88]. In Treemix it is up to the *user* to decide when to generalize and when to unfold. This is done by manual *annotation* of the subject program: every call is annotated as either "should always be unfolded" or "should never be unfolded" [Sestoft 86], and an expression may be annotated to indicate that its symbolic value should be generalized to a variable (called "dynamic rhs terms" in [Bondorf 88]).

6. Preprocessing

As argued in [Bondorf, Jones, Mogensen, & Sestoft 89], efficient self-application requires preprocessing in the form of *binding time analysis*. The purpose is to decide specialization time tests already in preprocessing. Since this removes work from the specializer, specialization can be performed more efficiently. More importantly, however, this means that self-application, specialization of the specializer, gives much better results (and thus better compilers). The reason is, shortly explained, that some important tests in the specializer being specialized can be decided due to the preprocessing of *its* static input, a subject program (e.g. an interpreter). Preprocessing adds information which was not otherwise present.

The information collected in preprocessing can be added to the program by *annotating* it. Treemix uses four preprocessing phases, each of which adds annotations to the subject program. Three of these phases are *abstract interpretations*, which abstract program specialization (symbolic evaluation) in different ways, depending on the desired information. Abstract interpretation gives a *safe approximation* to the computation it abstracts: the information computed by abstract interpretation may not be precise, but is always correct.

6.1 Call annotation analysis

Our first preprocessing phase abstracts call unfolding. The analysis predicts whether the user supplied call unfolding annotations possibly may result in disallowed case or equal tests (problem (1) of section 5.2). It assigns to every program variable a value from a three element lattice: $\bot \sqsubseteq \mathsf{P} \sqsubseteq \top$. The top value $\top$ abstracts function calls: if a variable is described by this value, it may possibly become bound to a call if some call to the function containing the variable is unfolded. If the variable is tested by case or equal, a disallowed form thus may result. It is therefore unsafe to unfold calls to the function containing the variable.

The bottom value $\bot$ abstracts values which definitely never will contain function calls. It is thus safe to test variables described by $\bot$. P abstracts values which are definitely not themselves function calls, but which may be pairs containing calls in the components. It is never safe to test a P value with equal, but in most cases case is safe: testing a pair against an atomic constant definitely gives a failing match, whereas testing it against a variable pair or else definitely gives a successful match. Thus, if a case only contains these pattern forms, it is always safe to test a pair since the case is guaranteed to be reduced away. It is only unsafe to test a P value against a non-atomic constant.

Only if all tests in a function are always safe, it can be guaranteed that calls to the function can safely be unfolded. Notice that this analysis does not at all deal with termination questions: there is no guarantee against infinite unfolding.

6.2 Binding time analysis

The abstract interpretation called binding time analysis was introduced for partial evaluation in [Jones, Sestoft, & Søndergaard 85]. It abstracts program specialization by abstracting away the static values. The simplest binding time domain is a two-point lattice: S for "definitely static" (or K for

"known") and D for "possibly dynamic" (or U for "possibly unknown") , ordered so that S $\sqsubseteq$ D. A binding time analysis for dealing with partially static structures is developed in [Mogensen 88]. It uses a domain with bottom value S and top value D. The values between describe *partially static* values. For instance, (S, D) describes a value which is a pair of a static and a possibly dynamic value. Compound values are ordered componentwise. To work with finite descriptions, *grammars* are introduced to handle recursive data structures.

Our analysis is an extension of Mogensen's: we distinguish between "definitely dynamic" (D) and "possibly dynamic" (U, unknown), and also between "atomic and static" (A) and "static" (S). The effect of our analysis is to assign to every variable a value from the domain

P means "partially static" (either definitely static, S, or a pair with arbitrary subparts). It is a compound domain described by grammars.

The binding time information is used to annotate case and equal to avoid the "cases" tests in the symbolic evaluation function *B* (section 5.1). For instance, if the tested variable in a case has the abstract value D, then the case is annotated so that the "isVar(v)" branch is always chosen. The abstract value U can be read as "no information"; if a tested variable is described by that abstract value, then the complete "cases" has to be performed.

6.3 Constructor analysis

The third abstract interpretation abstracts call unfolding (like the first one), but for a different purpose. The analysis is introduced due to problem (2) of section 5.2: call unfolding may necessitate expression restructuring to ensure that all tests occur in the beginning.

For all argument expressions to occurrences of the pairing operator, the analysis assigns a value $\bot$ or $\top$ ($\bot \sqsubseteq \top$). $\top$ means that the residual version of the argument expression (after symbolic evaluation including call unfolding) may contain tests. $\bot$ abstracts expressions which are guaranteed not to symbolically evaluate to expressions with tests, i.e. they evaluate to expressions of the syntactic form R. The information is used to annotate pairing operators. If both arguments are described with $\bot$, an annotation telling that post-restructuring is definitely not needed is added.

6.4 Unmodified functions analysis

This final analysis detects functions which definitely will appear unmodified in the residual program (except for renaming). Such functions need not be specialized but can simply be copied by the partial evaluator.

Functions for which all argument variables have the binding time value D are candidates for this. Other requirements are that calls to the function are never unfolded and that the function itself only calls other functions, which will appear unmodified (with calls that are not unfolded). A number of functions of this kind do appear in our partial evaluator itself, so the analysis is worthwhile for the aim of self-application.

7. Results

The partial evaluator has been implemented and successfully self-applied. The tables below summarize the various results. The first table shows run time figures; the specializer is referred to as mix, the preprocesser as pre. The figures for preprocessing include all 4 preprocessing phases. Three interpreters have been used in the experiments: MPint, an interpreter for a simple imperative "while" language with list data structures (described in [Sestoft 86]), lamint, an interpreter for a simple functional language with higher order functions (a subset of mini-ML [Kahn 87]) and finally sint, a Tree self-interpreter. The corresponding generated compilers are named MPcomp, lamcomp, and scomp respectively. scomp is a "self-compiler" (a source to source transformer): it "compiles" a Tree program into a Tree program. Finally, cogen is the compiler generator.

Tree has been implemented by compilation into Scheme. The programs have been run in Chez Scheme version 2.0.3 on a VAX 785, Unix 4.3 BSD. All run time figures are given in CPU seconds with one decimal; they exclude the time used

Run time figures

	time/ms	ratio
resultTree		
= L sint [sourceTree, inputTree]	4.5	54.2
= L targetTree inputTree	0.1	
resultMP		
= L MPint [sourceMP, inputMP]	32.2	19.5
= L targetMP inputMP	1.7	
resultlam		
= L lamint [sourcelam, inputlam]	11.1	2.0
= L targetlam inputlam	5.5	
targetTree		
= L mix [sint-ann, sourceTree]	12.3	9.6
= L scomp sourceTree	1.3	
targetMP		
= L mix [MPint-ann, sourceMP]	1.9	3.2
= L MPcomp sourceMP	0.6	
targetlam		
= L mix [lamint-ann, sourcelam]	1.1	1.7
= L lamcomp sourcelam	0.6	
scomp		
= L mix [mix-ann, sint-ann]	36.8	9.7
= L cogen sint-ann	3.8	
MPcomp		
= L mix [mix-ann, MPint-ann]	42.1	6.3
= L cogen MPint-ann	6.7	
lamcomp		
= L mix [mix-ann, lamint-ann]	58.2	4.0
= L cogen lamint-ann	14.7	
cogen		
= L mix [mix-ann, mix-ann]	326.3	5.8
= L cogen mix-ann	56.7	
sint-ann = L pre sint	6.1	
MPint-ann = L pre MPint	7.0	
lamint-ann = L pre lamint	4.7	
mix-ann = L pre mix	304.3	

for garbage collection (in the worst case 40% extra time, typically much less). The ratio numbers show how big the partial evaluation gain is: how much speedup we achieve by partial evaluation. More decimals than the ones given here have been used in the computation of the ratios. All figures have the usual uncertainty connected to CPU measures.

The figures compare well with those given in [Mogensen 88] (to the author's knowledge the only other existing fully self-applicable evaluator with partially static structures). It should be noted that no postprocessing is performed on output from mix; all postprocessing-like work is done by mix itself. Since postprocessing is not speeded up by self-application, the inclusion of postprocessing in mix gives worse self-application ratios than if it had been performed in a separate phase. The ratios are still very satisfactory, though.

Sizes

	words	ratio
sourceTree	367	1.0
targetTree	374	
sourceMP	67	3.3
targetMP	221	
sourcelam	21	2.5
targetlam	52	
sint	368	9.0
scomp	3308	
MPint	414	11.0
MPcomp	4535	
lamint	359	17.9
lamcomp	6425	
mix	4298	4.0
cogen	17219	
mix-ann	5123	

The best ratios by far are those for the self-interpreter. This is not surprising: Tree as well as Treemix were designed with particular attention to the self-interpreter case, cf. the discussion earlier. An interesting observation is that the figures for lamint are significantly worse than the other figures. The reason for this is that in lamint almost all variables get the binding time value U, thus leading to a badly annotated version. lamint uses higher order functions expressed indirectly in so-called named combinator form (with an explicit "@" operator for function application, cf. the combinator logic example in section 4.3), and this leads to many U binding time values. Efficient binding time analysis for higher order programming is a subject of current research [Mogensen 89].

The second table shows the sizes of various selected programs in source form; the figures are given as the number of words (tokens). In bytes, the largest program (cogen) has the size 111681. As with the run time figures, these are also comparable to those in [Mogensen 88]. Again, notice that the ratio for lamcomp is much worse than for the other compilers (this time a small ratio is desired); the figures for Tree/sint are generally the best. It is interesting that the ratios for cogen are significantly smaller (better) than those for the compilers.

8. Related work

[Bonacina 88] relates partial evaluation of term rewriting systems to equational (Knuth-Bendix) completion. She describes a very general partial evaluation algorithm based on the completion procedure; the approach is less operational than ours and does not address self-application.

[Consel 88] describes a compiler oriented partial evaluator for a subset of first order Scheme. Programs are written with special compile time and run time operations and are user annotated with flexible function annotations for deciding upon unfolding and specialization. In [Romanenko 88] a partial evaluator for a functional LISP-like language with REFAL data structures (strings rather than S-expressions) is described; he basically uses the principles from [Sestoft 86]. [Fuller & Abramsky 88] describes partial evaluation of Prolog.

[Turchin 86] describes *supercompilation*, a more general program transformation technique than partial evaluation. All decisions are taken "on the fly", including decisions on call unfolding and generalization. There is thus no preprocessing like our binding time analysis; the supercompiler has not been successfully self-applied.

[Launchbury 88] uses domain projections (retracts) to describe the division of data into static and dynamic parts.

9. Conclusion and issues

We have presented a fully self-applicable partial evaluator for an intermediate language for term rewriting systems. The language is strict (innermost reduction order), first order, and untyped. We have overcome the problems with partial evaluation of pattern matching by expressing it at a lower level of abstraction: pattern matching is translated into decision (matching) trees. Decision trees have a rather restrictive syntax to model term rewriting systems closely. This introduces new problems in partial evaluation, in particular in connection to call unfolding. We have implemented the partial evaluator and self-applied it with satisfactory results. Reasonably small and efficient compilers as well as a compiler generator have been generated.

It is conceivable that an idea similar to decision trees is useful for partial evaluation of Prolog: translate a set of clauses with the same predicate symbol in the clause heads into a decision tree with the clause bodies at the leaves. The decision tree branching would be decided by elementary unification operations such as unification of variables with constants, unification for decomposing structures, and unification of two variables (for dealing with repeated variables in clause heads).

Future work in partial evaluation should address higher order programming; this is not handled efficiently by present day self-applicable partial evaluators. Normal order (lazy) reduction also deserves attention; no self-applicable partial evaluator exists for a lazy language (at least to our knowledge). Finally, more work along the lines of [Jones 88] is needed for handling the in general undecidable problem of termination. The aim here is to develop algorithms for automatically finding safe generalization strategies that ensure termination.

Acknowledgements

I am most grateful to Torben Mogensen for many fruitful discussions on partial evaluation. Olivier Danvy has read drafts and has patiently answered my questions concerning the Scheme implementation of the system. Thanks also go to Charles Consel, Harald Ganzinger, Carsten Kehler Holst, Neil D. Jones, John Launchbury, and Peter Sestoft.

References

[Augustsson 85]
Lennart Augustsson: "Compiling pattern matching", *Conference on Functional Programming Languages and Computer Architecture* (ed. J.-P. Jouannaud), Nancy, France 1985, *Lecture Notes in Computer Science* **201**, 368-381, Springer-Verlag 1985.

[Baeten, Bergstra, & Klop 87]
J. C. M. Baeten, J. A. Bergstra, and J. W. Klop: "Term rewriting systems with priorities", *Rewriting Techniques and Applications* (ed. Pierre Lescanne), Bordeaux, France 1987, *Lecture Notes in Computer Science* **256**, 83-94, Springer-Verlag 1987.

[Bonacina 88]
Maria Paola Bonacina: "Partial evaluation in functional rewrite programming", *AICA Annual Conference*, Cagliari, Italy 1988.

[Bondorf 88]
Anders Bondorf: "Towards a self-applicable partial evaluator for term rewriting systems", in [PEMC 88].

[Bondorf, Jones, Mogensen, & Sestoft 89]
Anders Bondorf, Neil D. Jones, Torben Æ. Mogensen, and Peter Sestoft: "Binding time analysis and the taming of self-application", *submitted for publication.*

[Bulyonkov 88]
M. A. Bulyonkov: "A theoretical approach to polyvariant mixed computation", in [PEMC 88].

[Consel 88]
Charles Consel: "New insights into partial evaluation: the SCHISM experiment", *ESOP '88* (ed. Harald Ganzinger), Nancy, France 1988, *Lecture Notes in Computer Science* **300**, 236-247, Springer-Verlag 1988.

[Dershowitz 85]
Nachum Dershowitz: "Computing with rewrite systems", *Information and Control* **65**, 122-157, 1985.

[Ershov 82]
Andrei P. Ershov: "Mixed computation: potential applications and problems for study", *Theoretical Computer Science* **18**, 41-67, 1982.

[Fuller & Abramsky 88]
David A. Fuller and Samson Abramsky: "Mixed computation of Prolog programs", in [PEMC 88].

[Futamura 71]
Yoshihiko Futamura: "Partial evaluation of computing process — an approach to a compiler-compiler", *Systems, Computers, Controls* **2**, 5, 45-50, 1971.

[Hoffmann & O'Donnell 82]
Christoph M. Hoffmann and Michael J. O'Donnell: "Programming with equations", *ACM Transactions on Programming Languages and Systems* **4**, **1**, January 1982.

[Huet & Levy 79]
Gérard Huet and Jean-Jacques Levy: "Computations in Nonambiguous Linear Rewriting Systems", Technical report no. 359, INRIA, Rocquencourt, France, 1979.

[Huet & Oppen 80]
Gérard Huet and Derek C. Oppen: "Equations and rewrite rules, a survey", *Formal Language Theory, Perspectives and Open Problems* (ed. Ronald V. Book), 349-405, Academic Press, 1980.

[Jones 87]
Neil D. Jones: "Flow analysis of lazy higher-order functional programs", in *Abstract Interpretation of Declarative Languages* (eds. Samson Abramsky and Chris Hankin), Ellis Horwood Series in Computers and their Applications, 1987.

[Jones 88]
Neil D. Jones: "Re-examination of automatic program specialization", in [PEMC 88].

[Jones, Sestoft, & Søndergaard 85]
Neil D. Jones, Peter Sestoft, and Harald Søndergaard: "An experiment in partial evaluation: The generation of a compiler generator", *Rewriting Techniques and Applications* (ed. J.-P. Jouannaud), Dijon, France 1985, *Lecture Notes in Computer Science* **202**, 124-140, Springer-Verlag 1985.

[Jones, Sestoft, & Søndergaard 88]
Neil D. Jones, Peter Sestoft, and Harald Søndergaard: "Mix: a self-applicable partial evaluator for experiments in compiler generation", *LISP and Symbolic Computation* **1** 3/4, 1988.

[Kahn 87]
Gilles Kahn: "Natural Semantics", INRIA, Centre Sophia Antipolis, France, Rapport de Recherche No 601, 1987.

[Klop 87]
J. W. Klop: "Term rewriting systems: a tutorial", *Bulletin of the European Association for Theoretical Computer Science* **32**, 1987.

[Launchbury 88]
John Launchbury: "Projections for specialisation", in [PEMC 88].

[Mogensen 88]
Torben Æ. Mogensen: "Partially static structures in a self-applicable partial evaluator", in [PEMC 88].

[Mogensen 89]
Torben Æ. Mogensen: "Binding time analysis for higher order polymorphically typed languages", this volume.

[PEMC 88]
D. Bjørner, A. P. Ershov, and N. D. Jones (eds.): "Workshop on Partial Evaluation and Mixed Computation", Gl. Avernæs, Denmark, October 1987. North-Holland 1988.

[Peyton Jones 87]
Simon L. Peyton Jones: "The Implementation of Functional Programming Languages", *Prentice-Hall*, 1987, in particular ch. 5, 78-103: "Efficient compilation of pattern-matching" by Philip Wadler.

[Romanenko 88]
Sergei A. Romanenko: "A compiler generator produced by a self-applicable specialiser can have a surprisingly natural and understandable structure", in [PEMC 88].

[Sestoft 86]
Peter Sestoft: "The structure of a self-applicable partial evaluator", *Programs as Data Objects* (eds. Neil D. Jones and Harald Ganzinger), Copenhagen, Denmark 1985, *Lecture Notes in Computer Science* **217**, 236-256, Springer-Verlag 1986.

[Sestoft 88]
Peter Sestoft: "Automatic call unfolding in a partial evaluator", in [PEMC 88].

[Turchin 86]
Valentin F. Turchin: "The concept of a supercompiler", *ACM Transactions on Programming Languages and Systems* **8**, 3, July 1986.

[Turchin 88]
Valentin F. Turchin: "The algorithm of generalization", in [PEMC 88].