Partial Evaluation of Order-sorted Equational Programs modulo Axioms ⋆

María Alpuente[1], Angel Cuenca[1], Santiago Escobar[1], and José Meseguer[2]

[1] DSIC-ELP, Universitat Politècnica de València, Spain.
{alpuente,acuenca,sescobar}@dsic.upv.es
[2] University of Illinois at Urbana-Champaign, USA. meseguer@illinois.edu

Abstract. Partial evaluation (PE) is a powerful and general program optimization technique with many successful applications. However, it has never been investigated in the context of expressive rule-based languages like Maude, CafeOBJ, OBJ, ASF+SDF, and ELAN, which support: 1) rich type structures with sorts, subsorts and overloading; 2) equational rewriting modulo axioms such as commutativity, associativity–commutativity, and associativity–commutativity–identity. In this extended abstract, we illustrate the key concepts by showing how they apply to partial evaluation of expressive rule-based programs written in Maude. Our partial evaluation scheme is based on an automatic unfolding algorithm that computes term *variants* and relies on *equational* least general generalization for ensuring global termination. We demonstrate the use of the resulting partial evaluator for program optimization on several examples where it shows significant speed-ups.

1 Introduction

Partial evaluation (PE) is a semantics-based program transformation technique in which a program is specialized to a part of its input that is known statically (at *specialization* time) [7, 10]. Partial evaluation has currently reached a point where theory and refinements have matured, substantial systems have been developed, and realistic applications can benefit from partial evaluation in a wide range of fields that transcend by far program optimization.

Narrowing-driven PE (NPE) [4, 5] is a generic algorithm for the specialization of functional programs that are executed by *narrowing* [9, 12], an extension of rewriting where matching is replaced by unification. Essentially, narrowing consists of computing an appropriate substitution for a symbolic program call in such a way that the program call becomes reducible, and then reduce it: both the rewrite rule and the term can be instantiated. As in logic programming, narrowing computations can be represented by a (possibly infinite) finitely branching tree. Since narrowing subsumes both rewriting and SLD-resolution, it is complete in the sense of both functional programming (computation of normal forms) and logic programming (computation of answers). By combining the functional dimension of narrowing with the power of logic variables and unification, the NPE approach has better opportunities for optimization than the more standard partial evaluation of logic programs (also known as *partial deduction*, PD) and functional programs [5].

To the best of our knowledge, partial evaluation has never been investigated in the context of expressive rule-based languages like Maude, CafeOBJ, OBJ, ASF+SDF, and ELAN, which

⋆ This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grants TIN 2015-69175-C4-1-R and TIN 2013-45732-C4-1-P, and by Generalitat Valenciana under grant PROMETEOII/2015/013, and by NSF grant CNS-1319109.

support: 1) rich type structures with sorts, subsorts and overloading; and 2) equational rewriting modulo axioms such as commutativity, associativity–commutativity, and associativity–commutativity–identity. In this extended abstract, we illustrate the key concepts by showing how they apply to partial evaluation of expressive rule-based programs written in Maude. The key NPE ingredients of [4] have to be further generalized to corresponding (order–sorted) *equational* notions (*modulo* axioms): e.g., *equational unfolding, equational closedness, equational embedding, and equational abstraction*; and the associated partial evaluation techniques become more sophisticated and powerful.

Let us motivate the power of our technique by reproducing the classical specialization of a program parser w.r.t. a given grammar into a very specialized parser [10].

Example 1. Consider the following rewrite theory (written in Maude[3] syntax) that defines an elementary parser for the language generated by simple, right regular grammars. We define a symbol `_|_|_` to represent the parser configurations, where the first underscore represents the (terminal or non-terminal) symbol being processed, the second underscore represents the current string pending to be recognised, and the third underscore stands for the considered grammar. We provide two non-terminal symbols `init` and `S` and three terminal symbols 0, 1, and the finalizing mark `eps` (for ε, the empty string). These are useful choices for this example, but they can be easily extended to more terminal and non-terminal symbols. Parsing a string *st* according to a given grammar Γ is defined by rewriting the configuration (`init | st |` Γ) using the rules of the grammar (in the opposite direction) to incrementally transform *st* until the final configuration (`eps | eps |` Γ) is reached.

```
fmod Parser is
  sorts Symbol NSymbol TSymbol String Production Grammar Parsing .
  subsort Production < Grammar .
  subsort TSymbol < String .
  subsorts TSymbol NSymbol < Symbol .
  ops 0 1 eps : -> TSymbol . ops init S : -> NSymbol . op mt : -> Grammar .
  op __ : TSymbol String -> String [right id: eps].
  op _->_ : NSymbol TSymbol -> Production .
  op _->_._ : NSymbol TSymbol NSymbol -> Production .
  op _;_ : Grammar Grammar -> Grammar [assoc comm id: mt] .
  op _|_|_ : Symbol String Grammar -> Parsing .
  var E : TSymbol . vars N M : NSymbol . var  L : String . var G : Grammar .
  eq (N | eps | ( N ->  eps ) ; G) =  (eps | eps | ( N ->  eps ) ; G) [variant] .
  eq (N | E L | ( N -> E . M ) ; G) = (M | L   | ( N -> E . M ) ; G) [variant] .
endfm
```

Note that this Maude equational program theory contains several novel features that are *unknown land* for (narrowing-driven) partial evaluation: 1) a subsorting relation `TSymbol` `NSymbol` $<$ `Symbol`, and 2) an associative-commutative with identity symbol `_;_` for representing grammars (meaning that they are handled as a multiset of productions), together with the symbol `__` with right identity for the input string. The general case of the parser is defined by the second equation that, given the configuration (`N | E L |` Γ) where (`E L`) is the string to be recognized, searches for the grammar production (`N -> E . M`) in Γ to recognize symbol `E`, and proceeds to recognize `L` starting from the non-terminal symbol `M`.

[3] In Maude 2.7, only equations with the attribute `variant` are used by the folding variant narrowing strategy, which is the only narrowing strategy considered in this paper.

Note that the combination of subtypes and equational (algebraic) axioms allows for a very compact definition.

For example, given the following grammar Γ generating the language $(0)^*(1)^*$:

```
init -> eps    init -> 0 . init   init -> 1 . S    S -> eps    S -> 1 . S
```

the initial configuration $(\texttt{init} \mid 0\,0\,1\,1\,\texttt{eps} \mid \Gamma)$ is deterministically rewritten as $(\texttt{init} \mid 0\,0\,1\,1\,\texttt{eps} \mid \Gamma) \to (\texttt{init} \mid 0\,1\,1\,\texttt{eps} \mid \Gamma) \to (\texttt{init} \mid 1\,1\,\texttt{eps} \mid \Gamma) \to (\texttt{S} \mid 1\,\texttt{eps} \mid \Gamma) \to (\texttt{S} \mid \texttt{eps} \mid \Gamma) \to (\texttt{eps} \mid \texttt{eps} \mid \Gamma)$.

We can specialize our parsing program to the productions of the given grammar Γ by partially evaluating the input term $(\texttt{init} \mid \texttt{L} \mid \Gamma)$, where L is a logical variable of sort `String`. By applying our partial evaluator, we aim to obtain the specialized parsing equations:

```
eq init || eps   = eps || eps .      eq init || 1 = eps || eps .
eq init || 0 L   = init || L .       eq S || eps  = eps || eps .
eq init || 1 1 L = S || L .          eq S || 1 L  = S || L .
```

which get rid of the grammar Γ (and hence of costly *ACU-matching* operations) while still recognizing string *st* by rewriting the simpler configuration (`init` || *st*) to the final configuration (`eps` || `eps`). We have run some test on both the original and the specialized programs with an impressive improvement in performance, see Section 3.

Our contribution. In this extended abstract, we delve into the essential ingredients of a partial evaluation framework for order sorted equational theories that is able to cope with subsorts, subsort polymorphism, convergent rules (equations), and equational axioms. We base our partial evaluator on a suitably extended version of the general NPE procedure of [4], which is parametric w.r.t. the *unfolding rule* used to construct finite computation trees and also w.r.t. an *abstraction operator* that is used to guarantee that only finitely many expressions are evaluated. For unfolding we use *(folding) variant narrowing* [8], a novel narrowing strategy for convergent equational theories that computes *most general variants* modulo algebraic axioms and is efficiently implemented in Maude. For the abstraction we rely on the *(order-sorted) equational least general generalization* recently investigated in [2].

2 Specializing Equational Theories modulo Axioms

In this section, we introduce a partial evaluation algorithm for an equational theory decomposed as a triple $(\Sigma, B, \overrightarrow{E_0})$, where Σ is the signature, E_0 is a set of convergent (equations that are implicitly oriented as) rewrite rules and B is a set of commonly occurring axioms such as associativity, commutativity, and identity. Let us start by recalling the key ideas of the NPE approach. We assume the reader is acquainted with the basic notions of term rewriting, Rewriting Logic, and Maude (see, e.g, [6]).

2.1 The NPE Approach

Given a set R of rewrite rules and a set Q of program calls (i.e. input terms), the aim of NPE [4] is to derive a new set of rules R' (called a partial evaluation of R w.r.t. Q, or a partial evaluation of Q in R) which computes the same answers and irreducible forms (w.r.t. narrowing) than

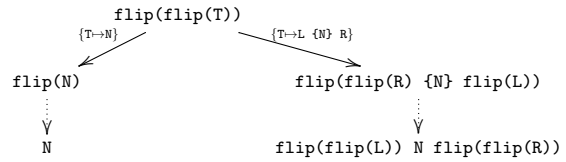

Fig. 1. Folding variant narrowing tree for the goal `flip(flip(T))`.

R for any term that t is inductively covered (*closed*) by the calls in Q. This means that every subterm in the leaves of the execution tree for t in R that can be narrowed (modulo B) in R can also be narrowed (modulo B) in R'. Roughly speaking, R' is obtained by first constructing a *finite* (possibly partial) narrowing tree for the input term t, and then gathering together the set of *resultants* $t\theta_1 \to t_1, \ldots, t\theta_k \to t_k$ that can be constructed by considering the leaves of the tree, say $t_1, \ldots, t_k$, and the computed substitutions $\theta_1, \ldots, \theta_k$ of the associated branches of the tree (i.e., a resultant rule is associated to each root-to-leaf derivation of the narrowing tree). Resultants perform what in fact is an n-step computation in R, with $n > 0$, by means of a single step computation in R'. The unfolding process is iteratively repeated for every narrowable subterm of $t_1, \ldots, t_k$ that is not covered by the root nodes of the already deployed narrowing trees. This ensures that resultants form a complete description covering all calls that may occur at run-time in R'.

Let us illustrate the classical NPE method with the following example that illustrates its ability to perform *deforestation* [13], a popular transformation that neither standard partial evaluation nor partial deduction can achieve [4]. Essentially, the aim of deforestation is to eliminate useless intermediate data structures, thus reducing the number of passes over data.

Example 2. Consider the following Maude program that computes the mirror image of a (non-empty) binary tree, which is built with the free constructor `_{_}_` that stores an element as root above two given (sub-)trees, its left and right children. Note that the program does not contain any equational attributes either for `_{_}_` or for the operation `flip` defined therein:

```
fmod FLIP-TREE is protecting NAT .
 sort NatTree . subsort Nat < NatTree .  vars R L : NatTree . var N : Nat .
 op _{_}_ : NatTree Nat NatTree -> NatTree .  op flip : NatTree -> NatTree .
 eq flip(N) = N [variant] .  eq flip(L {N} R) = flip(R) {N} flip(L) [variant] .
endfm
```

By executing the input term `flip(flip(T))` this program returns the original tree back, but it first computes an intermediate, mirrored tree `flip(T)` of `T`, which is then flipped again.

Let us partially evaluate the input term `flip(flip(T))` following the NPE approach, hence we compute the folding variant narrowing tree depicted[4] in Figure 1 for the term `flip(flip(T))`. This tree does not contain, altogether, uncovered calls in its leaves. Thus, we get the following residual program $\mathscr{R}'$ after introducing the new symbol `dflip`:

```
eq dflip(N) = N .          eq dflip(L {N} R) = dflip(L) {N} dflip(R) .
```

which is completely deforested, since the intermediate tree constructed after the first application of `flip` is not constructed in the residual program using the specialised definition of

[4] We show narrowing steps in solid arrows and rewriting steps in dotted arrows.

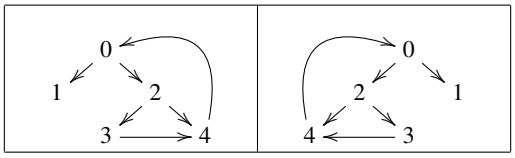

Fig. 2. Flipping a graph.

`dflip`. This is equivalent to the program generated by deforestation [13] but with a much better performance, see Section 3. Note that the fact that folding variant narrowing [8] ensures normalization of terms at each step is essential for computing the calls `flip(flip(R))` and `flip(flip(L))` that appear in the rightmost leaf of the tree in Figure 1, which are closed w.r.t. the root node of the tree.

When we specialize programs that contain sorts, subsorts, rules, and equational axioms, things get considerably more involved, as discussed in the following section.

2.2 Partial evaluation of convergent rules modulo axioms

Let us motivate the problem by considering the following variant of the `flip` function of Example 2 for (binary) graphs instead of trees.

Example 3. Consider the following Maude program for flipping binary graphs that are represented as multisets of nodes which may contain explicit, left and right, references (pointers) to their child nodes in the graph. We use symbol ♯ to denote an empty pointer. As expected, the `BinGraph` (set) constructor `_;_` obeys axioms of associativity, commutativity and identity (ACU). We consider a fixed set of identifiers `0`...`4`.

```
fmod GRAPH is sorts BinGraph Node Id Ref .
  subsort Node < BinGraph . subsort Id < Ref .
  op {___} : Ref Id Ref -> Node . op mt : -> BinGraph .
  op _;_ : BinGraph BinGraph -> BinGraph [assoc comm id: mt] .
  ops 0 1 2 3 4 : -> Id . --- Fixed identifiers
  op # : -> Ref . --- Void pointer
  var I : Id . vars R1 R2 : Ref . var BG : BinGraph .
endfm
```

We are interested in flipping a graph and define a function `flip` that takes a reference and a binary graph and returns the flipped graph.

```
op flip : BinGraph -> BinGraph .
eq [E1] : flip(mt) = mt  [variant] .
eq [E2] : flip({R1 I R2} ; BG) = {R2 I R1} ; flip(BG) [variant] .
```

We can represent the graph shown on the left-hand side of Figure 2 as the following term `BG` of sort BinGraph:

```
{ 1 0 2 } ; { # 1 # } ; { 3 2 4 } ; { # 3 4 } ; { # 4 0 }
```

By invoking `flip(BG)`, the graph shown on the right-hand side of Figure 2 is computed.

In order to specialize the previous program for the call `flip(flip(BG))`, we need several PE ingredients that have to be generalized to the corresponding (order–sorted) *equational* notions: (i) *equational closedness*, (ii) *equational embedding*, and (iii) *equational generalization*. In the following, we discuss some subtleties about these new notions gradually, through our graph-flipping running example.

2.3 Equational closedness and the generalized Partial Evaluation scheme

Roughly speaking, in order to compute a specialization for t in $(\Sigma, B, \overrightarrow{E_0})$, we need to start by constructing a finite (possibly partial) $(\overrightarrow{E_0}, B)$-narrowing tree for t using the folding variant narrowing strategy [8], and then extracting the specialized rules $t\sigma \Rightarrow r$ (resultants) for each narrowing derivation $t \leadsto_{\sigma, \overrightarrow{E_0}, B} r$ in the tree. However, in order to ensure that resultants form a complete description covering all calls that may occur at run-time in the final specialized theory, partial evaluation must rely on a parametric general notion of *equational Q-closedness* (modulo B) that is not a mere syntactic subsumption check (i.e., to be a substitution instance of some term in Q as in the partial deduction of logic programs), but recurses over the algebraic B-structure of the terms.

Definition 1 (Equational Closedness). *Let $(\Sigma, B, \overrightarrow{E_0})$ be an equational theory decomposition and Q be a finite set of Σ-terms. Assume the signature Σ splits into a set $\mathscr{D}_{E_0}$ of defined function symbols and a set $\mathscr{C}_{E_0}$ of constructor symbols (i.e., $\overrightarrow{E_0}, B$-irreducible), so that $\Sigma = \mathscr{D}_{E_0} \uplus \mathscr{C}_{E_0}$. We say that a Σ-term t is closed* modulo *B (w.r.t. Q and Σ), or B–closed, if $closed_B(Q,t)$ holds, where the predicate $closed_B$ is defined as follows:*

$$closed_B(Q,t) \Leftrightarrow \begin{cases} true & \text{if } t \text{ is a variable} \\ closed_B(Q,t_1) \wedge \ldots \wedge closed_B(Q,t_n) & \text{if } t = c(\overline{t_n}),\ c \in \mathscr{C}_{E_0},\ n \geq 0 \\ \bigwedge_{x \mapsto t' \in \theta} closed_B(Q,t') & \text{if } \exists q \in Q \text{ such that } q\theta =_B t \\ & \text{for some substitution } \theta \end{cases}$$

A set T of terms is closed modulo B (w.r.t. Q and Σ) if $closed_B(Q,t)$ holds for each t in T. A set R of rules is closed modulo B (w.r.t. Q and Σ) if the set $Rhs(R)$ consisting of the right-hand sides of all the rules in R is closed modulo B (w.r.t. Q and Σ).

Example 4. In order to partially evaluate the program in Example 3 w.r.t. the input term `flip(flip(BG))`, we set $Q = \{$`flip(flip(BG))`$\}$ and start by constructing the folding variant narrowing tree that is shown[5] in Figure 3.

When we consider the leaves of the tree, we identify two requirements for Q-closedness, with B being ACU: (i) $closed_B(Q,t_1)$ with $t_1 =$ `mt` and (ii) $closed_B(Q,t_2)$ with $t_2 =$ `{R1 I R2} ; flip(flip(BG'))`. The call $closed_B(Q,t_1)$ holds straightforwardly (i.e., it is reduced to *true*) since the `mt` leaf is a constant and cannot be narrowed. The second one $closed_B(Q,t_2)$ also returns true because `{R1 I R2}` is a flat constructor term and `flip(flip(BG'))` is a (syntactic) renaming of the root of the tree.

We now show an example that requires to use B-matching in order to ensure equational closedness modulo B.

[5] To ease reading, the arcs of the narrowing tree are decorated with the label of the corresponding equation applied at the narrowing step.

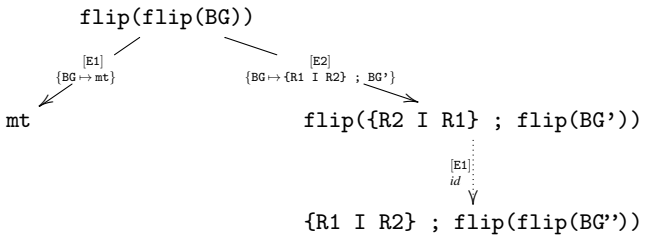

Fig. 3. Folding variant narrowing tree for the goal `flip(flip(BG))`.

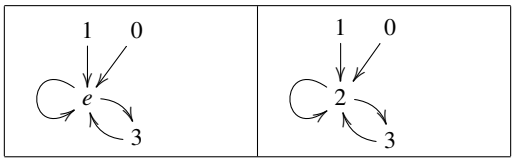

Fig. 4. Fixing a graph.

Example 5. Let us introduce a new sort `BinGraph?` to encode bogus graphs that may contain spurious nodes in a supersort `Id?` and homomorphically extend the rest of symbols and sorts. For simplicity, we just consider one additional constant symbol `e` in sort `Id?`.

```
sorts BinGraph? Id? Node? Ref? . subsort BinGraph Node? < BinGraph? .
subsort Node < Node? . subsort Id < Id? . subsort Ref Id? < Ref? .
op e : -> Id? .     op {___} : Ref? Id? Ref? -> Node? .
op _;_ : BinGraph? BinGraph? -> BinGraph? [ctor assoc comm id: mt] .
vars I I1 : Id . var I? : Id? . vars R1 R2 : Ref . vars R1? R2? : Ref? .
vars BG : BinGraph . var BG? : BinGraph? .
```

Let us consider a function `fix` that receives an extended graph `BG?`, an unwanted node `I?`, and a new content `I`, and traverses the graph replacing `I?` by `I`.

```
op fix : Id Id? BinGraph? -> BinGraph? .
eq [E3] : fix(I, I?, {R1? I? R2?} ; BG?) =
          fix(I, I?, {R1? I R2?} ; BG?) [variant] .
eq [E4] : fix(I, I?, {I? I1 R2?} ; BG?) =
          fix(I, I?, {I I1 R2?} ; BG?) [variant] .
eq [E5] : fix(I, I?, {R1? I1 I?} ; BG?) =
          fix(I, I?, {R1? I1 I} ; BG?) [variant] .
eq [E6] : fix(I, I?, BG) = BG [variant] .
```

For example, consider the following term `T` of sort BinGraph?:

```
{# 1 e} ; {e 0 #} ; {e e 3} ; {e 3 #}
```

that represents the graph shown on the left-hand side of Figure 4. By invoking `fix(2, e, T)`, we can fix the graph `T`, by computing the corresponding transformed graph shown on the right-hand side of Figure 4, where the unwanted node *e* has been replaced.

Now assume we want to specialize the above function `fix` w.r.t. the input term `fix(2, e, {R1 I R2} ; BG?)`, that is, a bogus graph with at least one non-spurious node `{R1 I`

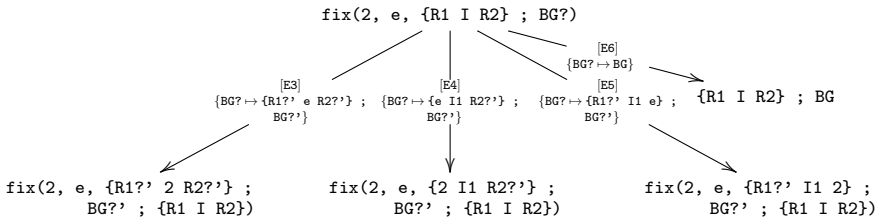

Fig. 5. Folding variant narrowing tree for the goal `fix(2, e, {R1 I R2} ; BG?)`.

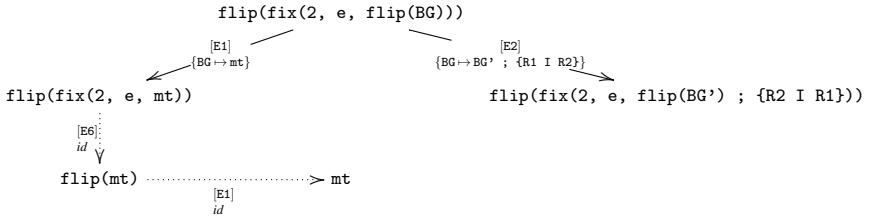

Fig. 6. Folding variant narrowing tree for the goal `flip(fix(2, e, flip(BG)))`.

`R2}` (non-spurious because of the sort of variable `I`). Following the proposed methodology, we set $Q = \{$`fix(2, e, {R1 I R2} ; BG?)`$\}$ and start by constructing the folding variant narrowing tree shown in Figure 5.

The right leaf `{R1 I R2} ; BG` is a constructor term and cannot be unfolded. The first two branches to the left of the tree are closed modulo *ACU* with the root of the tree in Figure 5. For instance, for the left leaf $t =$ `fix(2, e, {R1?' 2 R2?'} ; BG?' ; {R1 I R2})`, the condition $closed_B(Q,t)$ is reduced[6] to true because t is an instance (modulo *ACU*) of the root node of the tree, and the subterm $t' =$ `({R1?' 2 R2?'} ; BG?')` occurring in the corresponding (*ACU*-)matcher is a constructor term. The other branches can be proved *ACU*-closed with the tree root in a similar way.

Example 6 (Example 5 continued). Now let us assume that the function `flip` is extended to (bogus graphs of sort) `BinGraph?`, by extending equations `E1` and `E2` in the natural way. We specialize the whole program containing functions `flip` and `fix` w.r.t. input term `flip(fix(2, e, flip(BG)))`, that is, take a graph `BG`, flip it, then fix any occurrence of nodes `e`, and finally flip it again. The corresponding folding variant narrowing tree is shown in Figure 6. Unfortunately this tree does not represent all possible computations for (any *ACU*-instances of) the input term, since the narrowable redexes occurring in the tree leaves are not a recursive instance of the only partially evaluated call so far. That is, the term `flip(fix(2, e, flip(BG') ; {R2 I R1}))` of the rightmost leaf is not *ACU*-closed w.r.t. the root node of the tree. As in NPE, we need to introduce a methodology that recurses (modulo *B*) over the structure of the terms to augment the set of specialized calls in a controlled way, so as to ensure that all possible calls are covered by the specialization.

We are now ready to formulate the backbone of our partial evaluation methodology for equational theories that crystallize the ideas of the example above. Following the NPE approach, we define a generic algorithm (Algorithm 1) that is parameterized by:

[6] Note that this is only true because pattern matching modulo *ACU* is used for testing closedness.

1. a *narrowing relation* (with narrowing strategy $\mathscr{S}$) that constructs search trees,
2. an *unfolding rule*, that determines when and how to terminate the construction of the trees, and
3. an *abstraction operator*, that is used to guarantee that the set of terms obtained during partial evaluation (i.e., the set of deployed narrowing trees) is kept finite.

Algorithm 1 Partial Evaluation for Equational Theories

Require:
An equational theory $\mathscr{E} = (\Sigma, B, \overrightarrow{E_0})$ and a set of terms Q to be specialized in $\mathscr{E}$
Ensure:
A set Q' of terms s.t. $\text{UNFOLD}(Q', \mathscr{E}, \mathscr{S})$ is closed modulo B w.r.t. Q'

```
1: function EqNPE(ℛ,Q,𝒮)
2:     Q := Q↓_{E0→,B}
3:     repeat
4:         Q' := Q
5:         ℒ ← UNFOLD(Q',ℰ,𝒮)
6:         Q ← ABSTRACT(Q',ℒ,B)
7:     until Q' =_B Q
8:     return Q'
```

Informally, the algorithm proceeds as follows. Given the input theory $\mathscr{E}$ and the set of terms Q, the first step consists in applying the unfolding rule $\text{UNFOLD}(Q,\mathscr{E},\mathscr{S})$ to compute a finite (possibly partial) narrowing tree in $\mathscr{E}$ for each term t in Q, and return the set $\mathscr{L}$ of the (normalized) leaves of the tree. Then, instead of proceeding directly with the partial evaluation of the terms in $\mathscr{L}$, an abstraction operator $\text{ABSTRACT}(Q,\mathscr{L},B)$ is applied that properly combines each uncovered term in $\mathscr{L}$ with the (already partially evaluated) terms of Q, so that the infinite growing of Q is avoided. The abstraction phase yields a new set of terms which may need further specialization and, thus, the process is iteratively repeated while new terms are introduced.

The PE algorithm does not explicitly compute a partially evaluated theory $\mathscr{E}' = (\Sigma, B, E')$. It does so implicitly, by computing the set of partially evaluated terms Q' (that unambiguously determine E' as the set of resultants $t\sigma \Rightarrow r$ associated to the root-to-leaf derivations $t \leadsto_{\sigma,\overrightarrow{E_0},B} r$ in the tree, with t in Q'), such that the closedness condition for E' modulo B w.r.t. Q' is satisfied.

Example 7 (Example 5 continued). Now let us assume that the function `flip` is extended to (bogus graphs of sort) `BinGraph?`, updating equations `E1` and `E2` in the natural way. We specialize the whole program containing functions `flip` and `fix` w.r.t. input term `flip(fix(2, e, flip(BG)))`, that is, take a graph `BG`, flip it, then fix any occurrence of nodes `e`, and finally flip it again. The corresponding folding variant narrowing tree is shown in Figure 6. Unfortunately this tree does not represent all possible computations for (any *ACU*-instances of) the input term, since the narrowable redexes occurring in the tree leaves are not a recursive instance of the only partially evaluated call so far. That is, the term `flip(fix(2, e, flip(BG') ; {R2 I R1}))` of the rightmost leaf is not *ACU*-closed w.r.t. the root node of the tree. As in NPE, we need to introduce a methodology that recurses (modulo B) over the structure of the terms to augment in a controlled way the set of specialized calls, so as to ensure that all possible calls are covered by the specialization.

2.4 Equational homeomorphic embedding

Partial evaluation involves two classical termination problems: the so-called *local* termination problem (the termination of unfolding, or how to control and keep the expansion of the narrowing trees finite, which is managed by an unfolding rule), and the *global* termination (which concerns termination of recursive unfolding, or how to stop recursively constructing more and more narrowing trees).

For local termination, we need to define *equational homeomorphic embedding* by extending the standard notion of homeomorphic embedding with order-sorted information and reasoning modulo axioms. Embedding is a structural preorder under which a term t is greater than, i.e., it embeds, another term t', written as $t \triangleright t'$, if t' can be obtained from t by deleting some parts.

Embedding relations are very popular to ensure termination of *symbolic* transformations because, provided the signature is finite, for every infinite sequence of terms $t_1, t_2, \ldots$, there exist $i < j$ such that $t_i \trianglelefteq t_j$. Therefore, when iteratively computing a sequence $t_1, t_2, \ldots, t_n$, finiteness of the sequence can be guaranteed by using the embedding as a whistle [11]: whenever a new expression t_{n+1} is to be added to the sequence, we first check whether t_{n+1} embeds any of the expressions already in the sequence. If that is the case, we say that $\trianglelefteq$ whistles, i.e., it has detected (potential) non-termination and the computation has to be stopped. Otherwise, t_{n+1} can be safely added to the sequence and the computation proceeds. For instance, if we work modulo commutativity (C), we must stop a sequence where the term $u =$`s(s(X+Y)*(s(X)+0))` occurs after $v =$`s(X)*s(X+Y)`, since v embeds u modulo commutativity of $*$.

Definition 2 ((order-sorted) equational homeomorphic embedding). *Let $(\Sigma, B, \overrightarrow{E_0})$ be an equational theory decomposition. Consider the TRS Emb(Σ) that consists of all rewrite rules $f(X_1 : A_1, \ldots, X_n : A_n) \to X_i : A_i$ with $f : A_1, \ldots, A_n \to A$ in Σ and $i \in \{1, \ldots, n\}$. For terms u and v we write $u \triangleright_B v$ if $u \to^+_{Emb(\Sigma)/B} v'$ and v' is equal to v up to B-renaming (i.e. $v \overset{ren}{=}_B v'$ iff there is a renaming substitution σ such that $v =_B v'\sigma$). The relation $\trianglelefteq_B$ is called B–embedding (or embedding modulo B).*

By using this notion, we stop a branch $t \rightsquigarrow t'$ of a folding variant narrowing tree, if any narrowing redex of the leaf t' is embedded (modulo B) by the narrowing redex of a preceding term u in the branch, i.e., $u|_p \trianglelefteq_B t'|_q$.

Example 8 (Example 7 continued). Consider again the (partial) folding variant narrowing tree of Figure 6. The narrowing redex $t =$ `flip(fix(2, e, flip(BG')) ; {R2 I R1})` in the right branch of the tree embeds modulo *ACU* the tree root $u =$ `flip(fix(2, e, flip(BG)))`. Since the whistle $u \trianglelefteq_B t$ blows, the unfolding of this branch is stopped.

2.5 Equational abstraction via equational least general generalization

For global termination, PE evaluation relies on an abstraction operation to ensure that the iterative construction of a sequence of partial narrowing trees terminates while still guaranteeing that the desired amount of specialization is retained and that the equational closedness condition is reached. In order to avoid constructing infinite sets, instead of just taking the union of the set $\mathscr{L}$ of non-closed terms in the leaves of the tree and the set Q of specialized calls, the sets Q and $\mathscr{L}$ are *generalized*. Hence, the abstraction operation returns a safe approximation

A of $Q \cup \mathscr{L}$ so that each expression in the set $Q \cup \mathscr{L}$ is closed w.r.t. A. Let us show how we can define a suitable abstraction operator by using the notion of *equational least general generalization (lgg_B)* [2]. Unlike the syntactical, untyped case, there is in general no unique lgg_B in the framework of [2], due to both the order-sortedness and to the equational axioms. Instead, there is a finite, minimal and complete set of lgg_B's for any two terms, so that any other generalizer has at least one of them as a B-instance.

More precisely, given the current set of already specialized calls Q, in order to add a set T of new terms, the function $\text{ABSTRACT}^{\circlearrowleft}(Q,T,B)$ of Algorithm 1 is instantiated with the following function, which relies on the notion of *best matching terms* (BMT), a proper generalization of [1] that is aimed at avoiding loss of specialization due to generalization. Roughly speaking, to determine the best matching terms for t in a set of terms U w.r.t., B, $BMT_B(U,t)$, for each u_i in U, we compute the set W_i of lgg_B's of t and u_i, and select the subset M of minimal upper bounds of the union $\bigcup_i W_i$. Then, the term u_j belongs to $BMT_B(Q,t)$ if at least one lgg element in the corresponding W_j belongs to M.

Example 9. Let $t \equiv g(1) \oplus 1 \oplus g(Y)$, $U \equiv \{1 \oplus g(X), X \oplus g(1), X \oplus Y\}$, and consider B to consist of the associative-commutative (AC) axioms for $\oplus$. To compute the best matching terms for t in U, we first compute the sets of lgg_B's of t with each of the terms in U:

$$\begin{array}{l} W_1 = lgg_{AC}(\{g(1) \oplus 1 \oplus g(Y), 1 \oplus g(X)\}) = \{\langle\{Z \oplus 1, \{Z/g(1) \oplus g(Y)\}, \{Z/g(X)\}\rangle, \\ \qquad \langle Z \oplus g(W)\}, \{Z/1 \oplus g(1), W/Y\}, \{Z/1, W/X\}\rangle \\ W_2 = lgg_{AC}(\{g(1) \oplus 1 \oplus g(Y), X \oplus g(1)\}) = \{\langle\{Z \oplus g(1)\}, \{Z/g(1) \oplus g(Y)\}, \{Z/X\}\rangle\} \\ W_3 = lgg_{AC}(\{g(1) \oplus 1 \oplus g(Y), X \oplus Y)\}) = \langle\{Z \oplus W\}, \{Z/1, W/g(1) \oplus g(Y)\}, \{Z/X, W/Y\}\rangle \end{array}$$

Now, the set M of minimal upper bounds of the set $W_1 \cup W_2 \cup W_3$ is $M = \{Z \oplus 1, Z \oplus g(1)\}$ and thus we have: $BMT_{AC}(S,t) = \{1 \oplus g(X), X \oplus g(1)\}$.

Definition 3 (equational abstraction operator). *Let Q,T be two sets of terms. We define $abstract^{\circlearrowleft}(Q,T,B) = abs_B^{\circlearrowleft}(Q,T)$, where:*

$$\begin{cases} abs_B^{\circlearrowleft}(\ldots abs_B^{\circlearrowleft}(Q,t_1),\ldots,t_n) & \text{if } T \equiv \{t_1,\ldots,t_n\}, n > 0 \\ Q & \text{if } T \equiv \emptyset \text{ or } T \equiv \{X\}, \text{with } X \in \mathscr{X} \\ abs_B^{\circlearrowleft}(Q,\{t_1,\ldots,t_n\}) & \text{if } T \equiv \{t\}, \text{with } t \equiv c(t_1,\ldots,t_n),\ c \in \mathscr{C}_{E_0} \\ generalize_B(Q,Q',t) & \text{if } T \equiv \{t\}, \text{with } t \equiv f(t_1,\ldots,t_n),\ f \in \mathscr{D}_{E_0} \end{cases}$$

where $Q' = \{t' \in Q \mid root(t) = root(t') \text{ and } t' \trianglelefteq_B t\}$, and the function generalize *is:*

$$\begin{array}{l} generalize_B(Q,\emptyset,t) = Q \cup \{t\} \\ generalize_B(Q,Q',t) = Q \text{ if } t \text{ is } Q-closed \\ generalize_B(Q,Q',t) = abs_B^{\circlearrowleft}(Q \setminus BMT_B(Q',t), Q'' \downarrow_{\overrightarrow{E_0},B}) \end{array}$$

where $Q'' = \{l \mid q \in BMT_B(Q',t), \langle w, \{\theta_1,\theta_2\}\rangle \in lgg_B(\{q,t\}), x \in \mathscr{D}om(\theta_1) \cup \mathscr{D}om(\theta_2),\ l \in \{w, x\theta_1, x\theta_2\}\}$.

Example 10 (Example 8 continued). Consider again the (partial) folding variant narrowing tree of Figure 6 with the leaf $t =$ `flip(fix(2, e, flip(BG') ; {R2 I R1}))` in the right branch of the tree and the tree root $u =$ `flip(fix(2, e, flip(BG)))`. We apply the abstraction operator with $Q = \{u\}$ and $T = \{t\}$. Since t is operation-rooted, we call $generalize_B(Q,Q',t)$ with $Q' = Q$, which in turn calls $abs_{ACU}^{\circlearrowleft}(Q \setminus BMT_{ACU}(Q',t), Q'')$, with $BMT_{ACU}(Q',t) = Q$ and $Q'' = \{w,v\}$, where $w =$ `flip(fix(2, e, flip(Bg) ; Bg'))` is the only *ACU* least general generalization of u and t and $v =$ `{R2' I' R1'}`. Then the call

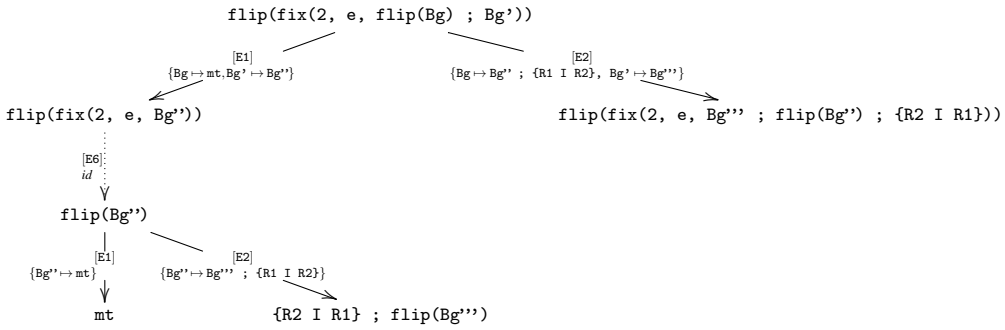

Fig. 7. Folding variant narrowing tree for the goal `flip(fix(2, e, flip(Bg) ; Bg'))`.

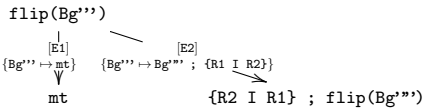

Fig. 8. Folding variant narrowing tree for the goal `flip(Bg''')`.

returns the set $\{w\}$. However, this means that the previous folding narrowing tree of Figure 6 is now discarded, since the previous set of input terms $Q = \{u\}$ is now replaced by $Q' = \{w\}$.

We start from scratch and the tree resulting for the new call w is showed in Figure 7. The right leaf embeds the root of the tree and is B-closed w.r.t. it. The left leaf `mt` is a constructor term. For the middle leaf $t'' =$ `{R2 I R1} ; flip(Bg''')` the whistle `flip(Bg'')` $\trianglelefteq_{ACU} t''$ blows and we stop the derivation. However, it is not B-closed w.r.t. w and we have to add it to the set Q', obtaining the new set of input terms $Q'' = \{w, \texttt{flip(Bg''')}\}$. The specialization of the call `flip(Bg''')` amounts constructing the narrowing tree of Figure 8, which is trivially ACU-closed w.r.t. its root.

Example 11 (Example 10 continued). Since the two trees in Figures 7 and 8 do represent all possible computations for (any ACU-instance of) $u =$ `flip(fix(2, e, flip(BG)))`, the partial evaluation process ends. Actually u is an instance of the root of the tree in Figure 7 with $\{\texttt{Bg'} \mapsto \texttt{mt}\}$ because of the identity axiom. The computed specialization is the set Q'''. Now we can extract the set of resultants $t\sigma \Rightarrow r$ associated to the root-to-leaf derivations $t \leadsto^{\vec{}}_{\sigma,\vec{E_0},B} r$ in the two trees, which yields:

```
eq flip(fix(2, e, flip(mt))) = mt .
eq flip(fix(2, e, flip({R1 I R2} ; BG'))) =
   flip(fix(2, e, flip(BG') ; {R2 I R1})) .
eq flip(fix(2, e, flip(mt) ; mt)) = mt .
eq flip(fix(2, e, flip(mt) ; Bg ; {R1 I R2})) = {R2 I R1} ; flip(Bg) .
eq flip(fix(2, e, flip({R1 I R2} ; Bg) ; Bg')) =
   flip(fix(2, e, flip(Bg) ; {R2 I R1} ; Bg')) .
eq flip(mt) = mt .
eq flip(Bg ; {R1 I R2}) = {R2 I R1} ; flip(Bg) .
```

The reader may have realized that the specialization call `flip(fix(2,e,flip(BG)))` should really return the same term `BG`, since the variable `BG` is of sort `BinGraph` instead of

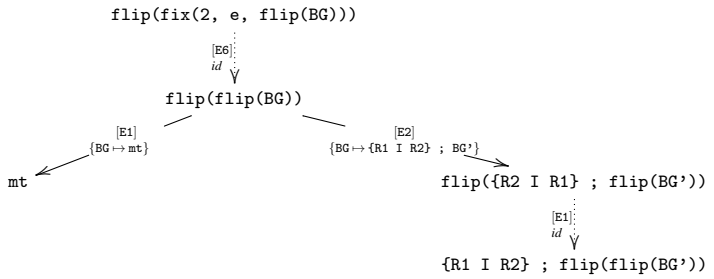

Fig. 9. Folding variant narrowing tree for the goal `flip(fix(2, e, flip(BG)))`.

`BinGraph?`, i.e., `flip(fix(2,e,flip(BG))) = BG`. The resultants above traverse the given graph and return the same graph. Though the code may seem inefficient, we have considered this example to illustrate the different stages of partial evaluation. The following example shows how a better specialization program can be obtained.

Example 12. Let us now consider a variant of function `fix` where its sort is declared as:

```
op fix : Id Id? BinGraph? -> BinGraph .
```

instead of

```
op fix : Id Id? BinGraph? -> BinGraph? .
```

Then, if we now specialize the call $t =$ `flip(fix(2,e,flip(BG)))` in the resulting mutated program, the narrowing tree for t is shown in Figure 9. The narrowing tree is B-closed w.r.t. the set of calls {`flip(fix(2, e, flip(BG)))`, `flip(flip(BG'))`} (normalized) root of the tree and leads to the following, optimal specialized program:

```
eq flip(fix(2,e,flip(mt))) = mt .
eq flip(fix(2,e,flip({R1 I R2} ; BG))) = {R1 I R2} ; flip(flip(BG)) .
eq flip(flip(mt)) = mt .
eq flip(flip({R1 I R2} ; BG)) = {R1 I R2} ; flip(flip(BG)) .
```

2.6 Post-processing renaming

The resulting partial evaluations might be further optimized by eliminating redundant function symbols and unnecessary repetition of variables. Essentially, we introduce a new function symbol for each specialized term and then replace each call in the specialized program by a call to the corresponding renamed function.

Example 13 (Example 12 continued). Consider the following independent renaming for the specialized calls: {`flip(flip(BG))` $\mapsto$ `dflip(BG)`,`flip(fix(2,e,flip(BG)))` $\mapsto$ `dflip-fix(BG)`}. The post-processing renaming derives the renamed program

```
eq dflip-fix(mt) = mt .
eq dflip(mt) = mt .
eq dflip-fix({R1 I R2} ; BG) = {R1 I R2} ; dflip(BG) .
eq dflip({R1 I R2} ; BG') = {R1 I R2} ; dflip(BG') .
```

Example 14. Consider again the elementary parser defined in Example 1 and the initial configuration `init | L` $\mid \Gamma$. Following the PE algorithm, we construct the two folding variant narrowing trees that are shown in Figures 10 and 11. Now all leaves in the tree are closed w.r.t. Q, and by applying the post-partial evaluation transformation with the independent renaming $\rho = \{$`init | L` $\mid \Gamma \mapsto$ `finit(L)`, `S | L` $\mid \Gamma \mapsto$ `fS(L)`, `eps | eps` $\mid \Gamma \mapsto$ `feps`$\}$, we get the following specialized program

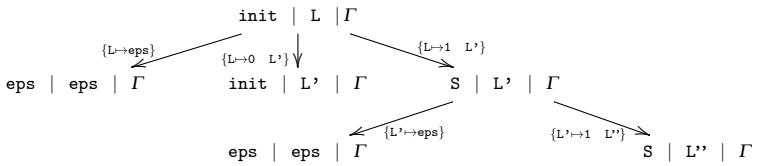

Fig. 10. Folding variant narrowing tree for the goal `init | L |`Γ.

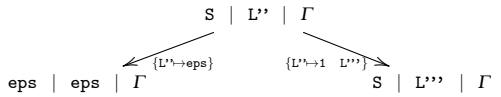

Fig. 11. Folding variant narrowing tree for the goal `S | L'' |` Γ.

```
eq finit(eps)   = feps .          eq finit(1) = feps .
eq finit(0 L)   = finit(L) .      eq fS(eps)  = feps .
eq finit(1 1 L) = fS(L) .         eq fS(1 L)  = fS(L) .
```

that is even more efficient and readable than the specialized program shown in the Introduction. Note that we obtain `finit(1 eps) = feps` but it is simplified to `finit(1) = feps` modulo identity.

3 Experiments and Conclusions

We have implemented the transformation framework presented in this paper. We do not yet have an automated tool where you can give both a Maude program and an initial call, and the tool returns the specialized program. However, all the independent components are already available and we have performed some experiments in a semi-automated way, i.e., we make calls to the different components already available without having a real interface yet: equational unfolding (by using folding variant narrowing already available in Maude; see [6]), equational closedness (we have implemented Definition 1 as a Maude program), equational embedding (we have implemented Definition 2 as a Maude program), and equational generalization and abstraction (we have implemented Definition 3 as a Maude program that invokes a Maude program defining the least general generalization of [2]).

Table 1 contains the experiments that we have performed using a MacBook Pro with an Intel Core i7 (2.5Ghz) processor and 8GB of memory and considering the average of ten executions for each test. These experiments are available at `http://safe-tools.dsic.upv.es/victoria`. We have considered the three Maude programs discussed in the paper: Parser (Example 1), Double-flip (Example 2), and Flip-fix (Example 3), and three sizes of input data: one hundred thousand elements, one million elements, and five million elements. Note that elements here refer to graph nodes for Double-flip and Flip-fix, and list elements for Parser. We have benchmarked three versions of each program on these data: original program, partially evaluated program (before post-processing renaming), and final specialization (with post-processing renaming). The relative speedups that are achieved thanks to specialization are given in the Improvement column(s) and computed as the percentage $100 \times (\mathsf{OriginalTime} - \mathsf{PETime})/\mathsf{OriginalTime}$. For all the examples, the partially evaluated

		Original	PE before renaming		PE after renaming	
Benchmark	**Data**	**Time (ms)**	**Time (ms)**	**Improvement**	**Time (ms)**	**Improvement**
Parser	100k	156	40	74,36	35	77,56
Parser	1M	12.599	418	96,68	361	97,13
Parser	5M	299.983	2.131	99,29	1.851	99,38
Double-flip	100k	177	155	12,43	86	51,41
Double-flip	1M	1.790	1.584	11,51	871	51,34
Double-flip	5M	8.990	8.006	10,95	4.346	51,66
Flip-fix	100k	212	188	11,32	151	28,77
Flip-fix	1M	2.082	1.888	9,32	1.511	27,43
Flip-fix	5M	10.524	9.440	10,30	7.620	27,59

Table 1. Experiments

program has a significant improvement in the execution time when compared to the original program, both with and without renaming, but more noticeable after renaming. Actually, matching modulo axioms such as associativity, commutativity, and identity are pretty expensive operations that are massively used in Maude, and can be drastically reduced after specialization (i.e., the Parser example moves from a program with ACU and Ur operators to a program without axioms).

Developing a complete partial evaluator for the entire Maude language requires to deal with some features not considered in this work, and to experiment with refined heuristics that maximize the specialization power. Future implementation work will focus on automating the entire PE process for a large subset of the language including conditional rules, memberships, and conditional equations. This, in turn, will necessitate some new developments in the Maude narrowing infrastructure. In this sense, advancing the present PE research ideas will be a significant driver of new symbolic reasoning features in Maude.

References

1. E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In *Proc. of SAS'98*. Springer LNCS 1503:262–277, 1998.
2. M. Alpuente, S. Escobar, J. Espert, and J. Meseguer. A Modular Order-Sorted Equational Generalization Algorithm. *Inf. and Comput.*, 235(0):98 – 136, 2014.
3. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM'97*, pp. 151–162. ACM, 1997.
4. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.
5. M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es, 1998.
6. M. Clavel, F. Durán, S Escobar, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (version 2.7)*. Available at `http://maude.cs.illinois.edu`. March 2015.
7. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany*. Springer LNCS 1110, 1996.
8. S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *J. Log. Algebr. Program.*, 81(7-8):898–928, 2012.
9. M. Fay. First Order Unification in an Equational Theory. In *CADE'79*, pages 161–167, 1979.
10. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
11. M. Leuschel. Improving Homeomorphic Embedding for Online Termination. In *Proc. of LOPSTR'98*. Springer LNCS 1559:199–218, 1990.

12. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.
13. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.