# Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation*

Germán Vidal

DSIC, Technical University of Valencia
Camino de Vera s/n, E-46022 Valencia, Spain
gvidal@dsic.upv.es

**Abstract.** Program slicing has been mainly studied in the context of imperative languages, where it has been applied to many software engineering tasks, like program understanding, maintenance, debugging, testing, code reuse, etc. This paper introduces the first forward slicing technique for multi-paradigm declarative programs. In particular, we show how program slicing can be defined in terms of online partial evaluation. Our approach clarifies the relation between both methodologies and provides a simple way to develop program slicing tools from existing partial evaluators.

## 1 Introduction

Essentially, program slicing [34] is a method for decomposing programs by analyzing their data and control flow. It has many applications in the field of software engineering (e.g., program understanding, maintenance, debugging, merging, testing, code reuse, etc). This concept was originally introduced by Weiser [33] in the context of imperative programs. Surprisingly, there are very few approaches to program slicing in the context of declarative programming (some notable exceptions are, e.g., [13, 21, 24, 25, 29]). Roughly speaking, a *program slice* consists of those program statements which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *program dependence graph* [10, 19] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards or forwards (from the slicing criterion), which is known as *backward* or *forward* slicing, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. A complete survey on program slicing can be found, e.g., in [30].

The main purpose of partial evaluation techniques is to specialize a given program w.r.t. part of its input data—hence it is also known as *program specialization*. The partially evaluated program will be (hopefully) executed more

* This work has been partially supported by CICYT TIC 2001-2705-C03-01, by the Generalitat Valenciana under grant CTIDIA/2002/205, and by the MCYT under grants HA2001-0059, HU2001-0019 and HI2000-0161.

efficiently since those computations that depend only on the known data are performed—at partial evaluation time—once and for all. Many *online* partial evaluation schemes follow a common pattern: given a program and a partial call, the partial evaluator builds a *finite* representation—generally a graph—of the possible executions of the partial call and, then, systematically extracts a *residual* program (the partially evaluated program) from this graph. This view of partial evaluation clearly shows the similarities with program slicing: both techniques should construct a finite representation of some program execution, usually with part (or none) of the input data.

In this paper, we present a forward slicing method based on (online) partial evaluation. While the construction of a graph representing some program execution is quite similar in both techniques, the extraction of the final program is rather different. Partial evaluation usually achieves its effects by compressing paths in the graph and by renaming expressions in order to remove unnecessary function symbols. Hence, partial evaluation constructs a *new*, residual program. In contrast, program slicing should preserve the structure of the original program: statements can be (totally or partially) deleted but new statements cannot be introduced. Following [12], partial evaluators can be classified into the following categories:

- *monovariant*: each function of the original program gives rise to (at most) one residual function,
- *polyvariant*: each function of the original program may give rise to one or more residual functions,
- *monogenetic*: each residual function stems from one function of the original program, and
- *polygenetic*: each residual function may stem from one or more functions of the original program.

According to this classification, forward slicing can be seen as a particular form of *monovariant* and *monogenetic* partial evaluation (in order to preserve a one-to-one relation between the functions of the original and residual programs).

Unfortunately, monovariant/monogenetic partial evaluation could be rather imprecise, thus resulting in an unnecessarily large residual program (i.e., slice). To overcome this problem, we introduce an extended operational semantics to perform partial evaluations, which helps us to preserve as much information as possible while maintaining the monovariant/monogenetic nature of the process. In order to center the discussion, we present our developments in the context of a multi-paradigm declarative language which integrates features from functional and logic programming (like, e.g., Curry [17] or Toy [22]).

The main contributions of this work are the following: (1) We define the first forward slicing technique for functional logic programs. Moreover, the application of our developments to pure (lazy) functional programs would be straightforward, since either the syntax and the underlying (online) partial evaluators (e.g., positive supercompilation [28]) share many similarities. (2) Our method is defined in terms of an existing partial evaluation scheme; therefore, it is easy to implement by adapting current partial evaluators. Furthermore, we do not need to consider separately static/dynamic slicing, since partial evaluation naturally accepts partially instantiated calls. (3) Our approach helps us to clarify the relation between (forward) slicing and (online) partial evaluation. Also, we discuss several possibilities to perform backward slicing by extending our developments.

This paper is organized as follows. In the next section we introduce some foundations for understanding the subsequent developments. Section 3 introduces the forward slicing technique. First, we recall the narrowing-driven approach to partial evaluation (Sect. 3.1); then we introduce an algorithm for computing program dependences by partial evaluation (Sect. 3.2); finally, we present our method to extract program slices (Sect. 3.3). Section 4 discusses two possible extensions of the scheme in order to perform backward slicing. Finally, we report in Sect. 5 an implementation of our program slicing tool and conclude in Sect. 6 with a comparison to related work. More details and proofs of all technical results can be found in [32].

## 2 Foundations

Recent proposals for multi-paradigm declarative programming (including features from the functional, logic and concurrent paradigms) consider inductively sequential rewrite systems [6] as *source programs* and a combination of needed narrowing [7] (the counterpart of call-by-name evaluation) and residuation as operational semantics [14]. In actual implementations, e.g., the PAKCS environment [15] for Curry [17], programs may also include a number of additional features: calls to external (built-in) functions, concurrent constraints, higher-order functions, overlapping left-hand sides, guarded expressions, etc. In order to ease the compilation of programs as well as to provide a common interface for connecting different tools working on source programs, a *flat representation* for programs has recently been introduced. This representation is based on the formulation of [16] to express pattern-matching by case expressions. The complete flat representation is called FlatCurry [15, 17] and is used as an intermediate language during the compilation of source programs.

In order to simplify the presentation, we will only consider the *core* of the flat representation. Extending the developments in this work to the remaining features is not difficult and, indeed, the implementation reported in Sect. 5 covers most of the additional features. The syntax of flat programs is summarized in Fig. 1, where $\overline{o_n}$ stands for the *sequence* of objects $o_1, \ldots, o_n$. We consider the following domains:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $x, y, z$ | $\in$ | $\mathcal{X}$ | (variables) | $a, b, c$ | $\in$ | $\mathcal{C}$ | (constructor symbols) |
| $f, g, h$ | $\in$ | $\mathcal{F}$ | (defined functions) | $e_1, e_2, \ldots$ | $\in$ | $\mathcal{E}$ | (expressions) |
| $t_1, t_2, \ldots$ | $\in$ | $\mathcal{T}$ | (terms) | $v_1, v_2, \ldots$ | $\in$ | $\mathcal{V}$ | (values) |

The only difference between *terms* and *expressions* is that the latter may contain case expressions. We say that a term is *operation-rooted* (resp. *constructor-rooted*) if it has a defined function symbol (resp. a constructor symbol) at the outermost position. *Values* are terms in head normal form, i.e., variables or

| | | | | |
|---|---|---|---|---|
| $\mathcal{R} ::= D_1 \ldots D_m$ | (program) | $t ::= x$ | | (variable) |
| $D ::= f(\overline{x_n}) = e$ | (rule) | | $\mid\ c(\overline{t_n})$ | (constructor call) |
| $e ::= t$ | (term) | | $\mid\ f(\overline{t_n})$ | (function call) |
| $\mid\ \mathit{case}\ x\ \mathit{of}\ \{\overline{p_m \to e_m}\}$ | (rigid case) | $p ::= c(\overline{x_n})$ | | (flat pattern) |
| $\mid\ \mathit{fcase}\ x\ \mathit{of}\ \{\overline{p_m \to e_m}\}$ | (flexible case) | | | |

**Fig. 1.** Syntax of Flat Programs

constructor-rooted terms. A program $\mathcal{R}$ consists of a sequence of function definitions $D$ such that each function is defined by a single rule whose left-hand side contains only different variables as parameters. The right-hand side is an expression $e$ composed by variables, constructors, function calls, and case expressions for pattern-matching. The general form of a case expression is:[1]

$$(f)\mathit{case}\ x\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_m(\overline{x_{n_m}}) \to e_m\}$$

where $x$ is a variable, $c_1, \ldots, c_m$ are different constructors of the type of $x$, and $e_1, \ldots, e_m$ are expressions (possibly containing nested $(f)\mathit{case}$'s). The variables $\overline{x_{n_i}}$ are local variables which occur only in the corresponding subexpression $e_i$. The difference between *case* and *fcase* only shows up when the argument, $x$, is a free variable (within a particular computation): *case* suspends—which corresponds to *residuation*, i.e., pure functional reduction—whereas *fcase* nondeterministically binds this variable to a pattern in a branch of the case expression—which corresponds to either narrowing [7] and driving [31]. Note that our functional logic language mainly differs from typical (lazy) functional languages in the presence of flexible case expressions.

*Example 1.* Consider the well-known function `app` to concatenate two lists. It can be defined in the flat representation as follows:[2]

```
app x y  =  case x of { []       → y ;
                        (z : zs) → z : app zs y }
```

where [ ] denotes the empty list and `x : xs` a list with first element `x` and tail `xs`.

An automatic transformation from source (inductively sequential [6]) programs to flat programs is introduced in [16]. Translated programs always fulfill the following restrictions: case expressions in the right-hand sides of program rules appear always in the outermost positions (i.e., there is no case expression inside a function or constructor call) and all case arguments are variables, thus the syntax of Fig. 1 is general enough for our purposes. We shall assume these restrictions on flat programs in the following.

[1] We write $(f)\mathit{case}$ for either *fcase* or *case*.

[2] Although we consider in this work a first-order representation, we use a curried notation in concrete examples (as it is common practice in functional languages).

## 3 Forward Slicing

This section presents a forward slicing technique based on (online) partial evaluation. In our context, any program expression may play the role of *slicing criterion*. Therefore, we do not need to distinguish between dynamic and static slicing, it only depends on the degree of instantiation of the slicing criterion. Given a program and a (possibly incomplete) call—the slicing criterion—, we return a *program slice* containing those parts of the original program which are reachable from the slicing criterion, i.e., which are needed to evaluate the slicing criterion. Functions in the slice should belong to the original program, although we accept the removal of alternatives in case expressions and the elimination of (unnecessary) function calls.

*Example 2.* Let us consider the following simple program:

```
foo x y z = fst (len (app x y), snd z)
len x     = case x of { [] → Z; (y:ys) → Succ (len ys) }
app x y   = case x of { [] → y; (z:zs) → z:app zs y }
fst (x,y) = x
snd (x,y) = y
```

where natural numbers are build from constructors `Z` and `Succ`. Function "`foo`" computes the length of the list resulting from the concatenation of its two first arguments. The unnatural definition of this function will be useful to illustrate the effects of program slicing. Standard functions "`len`", "`app`", "`fst`", and "`snd`" return, respectively, the length of a list, the concatenation of two lists, the first element of a tuple, and the second element of a tuple. Let us consider that the slicing criterion is the expression "`foo [] y z`". Then the computed slice should be as follows:

```
foo x y z = fst (len (app x y), ⊤)
len x     = case x of { [] → Z; (y:ys) → Succ (len ys) }
app x y   = case x of { [] → y }
fst (x,y) = x
```

Here, the second alternative of the case expression in the right-hand side of function "`app`" has been removed, since it is not needed to execute the slicing criterion. Also, the evaluation of the call to "`snd`" in the right-hand side of function "`foo`" is not needed—the outermost function, "`fst`", only demands the evaluation of the first component of the tuple—and, thus, it has been replaced by $\top$, a special symbol denoting that some subterm is missing due to the slicing process. Since function "`snd`" is no longer necessary, its definition has been completely deleted. Note that this slice could not be constructed by using a simple graph of functional dependencies (e.g., function "`snd`" depends on function "`foo`" but it does not appear in the computed slice).

Usually, slicing criteria are program calls whose execution traces we are interested in (e.g., to correct a bug or to extract a program fragment which we want to reuse in another context).

As discussed in the introduction, our developments rely on the fact that forward slicing can be regarded as a form of monovariant/monogenetic partial evaluation. This requirement is necessary in order to ensure that there is a one-to-one relation between the functions of the original and residual programs, which is crucial to produce a fragment of the original functions rather than a specialized version. Basically, the following two points should be taken into account:

- Since monovariant/monogenetic partial evaluation propagates information poorly, we need a carefully designed operational mechanism which avoids the loss of information (i.e., program dependences) as much as possible.
- The extraction of residual rules should be modified in order to ensure that the residual program is always a *fragment* of the original one.

The rest of this section introduces our program slicing technique based on an appropriate extension of a monovariant/monogenetic partial evaluator. We proceed in a stepwise manner: first, we recall an online partial evaluation scheme; then we modify the kernel of this algorithm in order to compute program dependences; finally, we present a method to construct the desired program slice from the computed program dependences.

### 3.1 Monovariant/Monogenetic Partial Evaluation

In this section, we recall the main algorithm of narrowing-driven partial evaluation [3, 4]. Essentially, it proceeds by iteratively unfolding a set of function calls, testing the *closedness* of the unfolded expressions, and adding to the current set those calls (in the derived expressions) which are not closed. This process is repeated until all the unfolded expressions are closed, which guarantees the correctness of the transformation process [5], i.e., that the resulting set of expressions covers all the possible computations for the initial call. This iterative style of performing partial evaluation was first described by Gallagher [11] for the partial evaluation of logic programs. The computation of a closed set of expressions can be regarded as the construction of a graph with all the program points which are reachable from the initial call. Intuitively, an expression is *closed* whenever its maximal operation-rooted subterms (function calls) are instances of the already partially evaluated terms. Formally, the closedness condition is defined as follows:

**Definition 1.** *Let $E$ be a finite set of expressions. We say that an expression $e$ is closed w.r.t. $E$ (or $E$-closed) iff one of the following conditions hold:*

- *$e$ is a variable;*
- *$e = c(e_1, \ldots, e_n)$ is a constructor call and $e_1, \ldots, e_n$ are recursively $E$-closed;*
- *$e = (f)case\ e'\ of\ \{\overline{p_k \to e_k}\}$ is a case expression and $e', e_1, \ldots, e_k$ are recursively $E$-closed;*
- *$e$ is operation-rooted, there is an expression $e' \in E$, a matching substitution $\sigma$, with $e = \sigma(e')$, and for all $x \mapsto e'' \in \sigma$, $e''$ is recursively $E$-closed.*

**Input:** a program $\mathcal{R}$ and a term $t$
**Output:** a residual program $\mathcal{R}'$
**Initialization:** $i := 0;\ E_0 := \{t\}$
**Repeat**
  $E' := \mathit{unfold}(E_i, \mathcal{R})$;
  $E_{i+1} := \mathit{abstract}(E_i, E')$;
  $i := i + 1$;
**Until** $E_i = E_{i-1}$ (modulo renaming)
**Return:**
  $\mathcal{R}' := \mathit{build\_residual\_program}(E_i, \mathcal{R})$

**Fig. 2.** Narrowing-Driven Partial Evaluation Procedure

The basic partial evaluation procedure is shown in Fig. 2. The operator *unfold* takes a program and a set of expressions, computes a *finite* set of (possibly incomplete) finite derivations, and returns the set of derived expressions. Function *abstract* is used to properly add the new expressions to the current set of (to be) partially evaluated expressions. The main loop of the algorithm can be seen as a *pre-processing* stage whose aim is to find a closed set of expressions. Note that no residual rules are actually constructed during this phase. Only when a closed set of expressions is eventually found, residual rules are built (usually, by applying one more time the unfolding operator, followed by a post-processing of renaming and some post-unfolding transformations).

Basically, a monovariant/monogenetic partial evaluation algorithm can be designed from the procedure in Fig. 2 by imposing several restrictions:

1. Expressions in the current set should be operation-rooted terms without nested function calls (i.e., of the form $f(\overline{t_n})$, where $f$ is a defined function symbol and $t_1, \ldots, t_n$ are constructor terms). This is necessary to ensure that partial evaluation is monogenetic.
2. The unfolding operator should perform only a one-step evaluation of each call. This condition is required to guarantee that no reachable function is hidden by the unfolding process.
3. Finally, the abstraction operator should ensure that the current set of terms contains at most one term for each function symbol. In this way, we enforce the monovariant nature of the partial evaluation process.

A trivial partial evaluator fulfilling the above restrictions could proceed by *flattening* all terms containing nested function symbols and by replacing those terms rooted by the same function symbol with some appropriate generalization (e.g., their *most specific generalization*). For instance, a term of the form "`len (app [ ] [ ])`" would be replaced by the terms "`len y`" and "`app [ ] [ ]`". However, this naive treatment would imply a serious loss of precision, e.g., the fact that `len` is only called with the result of "`app [ ] [ ]`" (an empty list).

To avoid this loss of precision, we drop the first restriction above, i.e., we consider arbitrary operation-rooted terms (possibly) containing nested function calls. Also, we extend the partial evaluation mechanism in order to work on states rather than on expressions. States have the form $\langle e, S\rangle$, where $e$ is the expression to be evaluated and $S$ is the *stack* (a list) which represents the current "evaluation context".[3] The empty stack is denoted by $[\,]$. An important property of our flat language is that it evaluates function calls lazily: an expression containing nested function calls is evaluated by first unfolding the outermost function; inner function calls are only evaluated *on demand*, i.e., when they appear as the argument of some case expression. For instance, "`len (app [] [])`" is unfolded to "`case (app [] []) of {...}`"; then the evaluation of function "`len`" cannot continue until the inner call to "`app`" is reduced to a value. Unfortunately, this interleaved evaluation is problematic in our context since it would give rise to a polygenetic partial evaluation. In contrast, we should perform a *complete one-step unfolding* of each function call separately (i.e., a function unfolding followed by the reduction of all the case structures in the unfolded expression). Here, the stack becomes relevant in order to store outer function calls until a complete one-step unfolding is possible. The next section introduces an extended semantics which is appropriate to deal with states.

### 3.2 Computing Program Dependences

In our context, the computation of program dependences boils down to computing the set of functions which are *reachable* from the slicing criterion, i.e., the functions which are *needed* to evaluate the slicing criterion. For this purpose, we introduce the extended operational semantics of Fig. 3. Let us briefly explain the rules of this operational semantics. Rule select is used to select the appropriate branch and continue with the evaluation of this branch. When the argument of a case expression is a free variable, rule guess is used to non-deterministically choose one alternative and continue with the evaluation of this branch; thus, the resulting calculus becomes non-deterministic as well. Rule flatten is used to avoid the unfolding of those (operation-rooted) terms whose unfolding would demand the evaluation of some inner call. In this case, we delay the function unfolding and continue evaluating the demanded inner call. Auxiliary function *flat* is used to flatten these states. Here, we use subscripts in the arrows to indicate the application of some concrete rule(s). Function *flat* proceeds as follows: When the expression in the input state can be reduced by using rules select and guess to a case expression with a function call in the argument position (which is thus *demanded*), function *flat* returns a new state whose first component is the demanded call, $g(\overline{t'_m})$, and whose stack is augmented by adding a new pair $(f(\overline{t_n})[g(\overline{t'_m})/x], x)$. Here, $f(\overline{t_n})[g(\overline{t'_m})/x]$ denotes the term obtained from $f(\overline{t_n})$ by replacing the selected occurrence of the inner call, $g(\overline{t'_m})$, with a fresh variable $x$. This pair contains all the necessary information to reconstruct the original expression once the inner call is evaluated to a value (in rule replace). Rule fun performs a simple function unfolding when rule flatten does not apply, i.e., when

[3] Similar operational semantics which make use of a stack can be found in [2, 27].

$$
\begin{array}{llrcll}
(\mathsf{select}) & \langle\ (f)case\ c(\overline{t_n})\ of\ \{\overline{p_k \to e_k}\}, & S\ \rangle & \Longrightarrow & \langle\ \rho(e_i), & S\ \rangle \\
& \text{if } p_i = c(\overline{x_n}) \text{ and } \rho = \{\overline{x_n \mapsto t_n}\} \text{ for some } i \in \{1,\ldots,k\} & & & & \\
(\mathsf{guess}) & \langle\ (f)case\ x\ of\ \{\overline{p_k \to e_k}\}, & S\ \rangle & \Longrightarrow & \langle\ \rho(e_i), & S\ \rangle \\
& \text{if } \rho = \{x \mapsto p_i\} \text{ for all } i = 1,\ldots,k & & & & \\
(\mathsf{flatten}) & \langle\ f(\overline{t_n}), & S\ \rangle & \Longrightarrow & \langle\ g(\overline{t'_m}), & S^f\ \rangle \\
& \text{if } \mathit{flat}(f(\overline{t_n}), S) = \langle g(\overline{t'_m}), S^f \rangle & & & & \\
(\mathsf{fun}) & \langle\ f(\overline{t_n}), & S\ \rangle & \Longrightarrow & \langle\ \rho(e), & S\ \rangle \\
& \text{if } \mathit{flat}(f(\overline{t_n}), S) = \bot,\ f(\overline{x_n}) = e \in \mathcal{R},\ \overline{x_n} \text{ are fresh, and } \rho = \{\overline{x_n \to t_n}\} & & & & \\
(\mathsf{replace}) & \langle\ v, & (f(\overline{t_n}), x) : S\ \rangle & \Longrightarrow & \langle\ \rho(f(\overline{t_n})), & S\ \rangle \\
& \text{if } v \text{ is a value and } \rho = \{x \mapsto v\} & & & &
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{where}\ \ \mathit{flat}(\langle f(\overline{t_n}), S\rangle) & = & \mathsf{if}\ \rho(e) \Longrightarrow^*_{\underline{\mathsf{select/guess}}} (f)case\ g(\overline{t'_m})\ of\ \{\ \ldots\ \} \\
& & \qquad \mathsf{then}\ \ \langle g(\overline{t'_m}), (f(\overline{t_n})[g(\overline{t'_m})/x], x) : S\rangle \\
& & \qquad \mathsf{else}\ \ \bot \\
& & \mathsf{with}\ f(\overline{x_n}) = e \in \mathcal{R},\ \overline{x_n}\ \mathsf{fresh,\ and}\ \rho = \{\overline{x_n \to t_n}\}
\end{array}
$$

**Fig. 3.** Extended Operational Semantics

function *flat* returns $\bot$. Finally, rule replace allows us to retake the evaluation of some delayed function call once the demanded inner call is reduced to a value.

The extended operational semantics behaves almost identically to the standard semantics for flat programs of [16]. There are, though, two slight differences:

- In the standard semantics, rigid case expressions with a free variable in the argument position *suspends*. In our case, rule guess proceeds with their evaluation as if they were flexible. This is motivated by the fact that we may have *incomplete* information; hence, in order to be on the safe side (and do not miss any reachable function), we should consider all the alternatives of rigid case expressions.
- The order of evaluation is changed. In our extended semantics, we delay those function unfoldings which cannot be followed by the reduction of all the case expressions in the corresponding right-hand side.

In spite of these differences, both calculi trivially produce the same results for input expressions involving no suspension. The relevance of the extended semantics stems from the fact that computations can now be split into a number of consecutive sequences of steps of the form:

$$
\Longrightarrow^*_{\mathsf{flatten}} \underbrace{\Longrightarrow_{\mathsf{fun}}\Longrightarrow^*_{\mathsf{select/guess}}}_{seq\_1} \Longrightarrow^*_{\mathsf{replace}}\Longrightarrow^*_{\mathsf{flatten}} \underbrace{\Longrightarrow_{\mathsf{fun}}\Longrightarrow^*_{\mathsf{select/guess}}}_{seq\_2} \Longrightarrow^*_{\mathsf{replace}} \cdots
$$

where each subsequence, *seq_i*, represents a complete one-step evaluation of some function call. From these sequences, a monogenetic/monovariant partial evaluation scheme can easily be defined (and, thus, our program slicing technique). Let

**Input:** a program $\mathcal{R}$ and an operation-rooted term $t$
**Output:** a set of states $\mathcal{S}$
**Initialization:** $i := 0$; $\mathcal{S}_0 := \{\langle t', S\rangle\}$, where $\langle t, [\,]\rangle \Longrightarrow^*_{\mathsf{flatten}} \langle t', S\rangle \not\Longrightarrow_{\mathsf{flatten}}$
**Repeat**
$\mathcal{S}' := \mathit{unfold}(\mathcal{S}_i, \mathcal{R})$;
$\mathcal{S}_{i+1} := \mathit{abstract}(\mathcal{S}_i, \mathcal{S}')$;
$i := i + 1$;
**Until** $\mathcal{S}_i = \mathcal{S}_{i-1}$ (modulo renaming)
**Return:** $\mathcal{S} := \mathcal{S}_i$

**Fig. 4.** Computation of Reachable Program Points

us note that our operational semantics could also be a good candidate to develop alternative approaches for computing program dependences (e.g., to develop a dependency graph analysis based on abstract interpretation).

The algorithm of Fig. 2 is now slightly modified in order to work with states. The new algorithm (Fig. 4) does not compute a residual program but the set of states which are reachable from the initial call; note that they are equivalent to the final set of *closed* terms computed by the algorithm of Fig. 2 (except for the fact that terms are represented by states). The new algorithm starts by flattening the initial term in order to ensure that a complete one-step unfolding can be performed. We now tackle the definition of appropriate unfolding and abstraction operators. Our one-step unfolding operator is defined as follows:

$$\mathit{unfold}(\mathcal{S}) = \bigcup_{s \in \mathcal{S}} \mathit{unf}(s)$$

where

$$\mathit{unf}(\langle t, S\rangle) = \{\langle t', S\rangle \mid \langle t, S\rangle \Longrightarrow_{\mathsf{fun}} \langle t'', S\rangle \Longrightarrow^*_{\mathsf{select/guess}} \langle t', S\rangle \not\Longrightarrow_{\mathsf{select/guess}}\}$$

This unfolding operator always performs a complete one-step unfolding of each input expression. The associated stack $S$ remains unchanged since only rules $\mathsf{flatten}$ and $\mathsf{replace}$ can modify the current stack. Function *unf* returns a *set* of derived states because of the non-determinism of the underlying operational semantics.

Before defining our abstraction operator, we need the following auxiliary notion. States returned by the unfolding operator and, then, reduced by rules $\mathsf{replace}$ and $\mathsf{flatten}$ are called *flattened* states. Formally, let $s$ be a state returned by operator *unfold* with $s \Longrightarrow^*_{\mathsf{replace/flatten}} s' \not\Longrightarrow$. Then $s'$ is called a *flattened* state. Flattened states have a particular form, as stated by the following lemma:

**Lemma 1.** *Let $s$ be a flattened state. Then $s$ has the form $\langle v, [\,]\rangle$, where $v$ is a value, or $\langle f(\overline{t_n}), S\rangle$, where $f(\overline{t_n})$ is an operation-rooted term.*

In order to add new states to the current set of states, our abstraction operator proceeds as follows:

$$\mathit{abstract}(\mathcal{S}, \{s_1, \ldots, s_n\}) = \mathit{abs}(\mathit{abs}(\ldots \mathit{abs}(\mathcal{S}, s'_1) \ldots, s'_{n-1}), s'_n)$$

where:

$$s_i \Longrightarrow^*_{\mathsf{replace/flatten}} s'_i \not\Longrightarrow_{\mathsf{replace/flatten}} \quad (\text{for all } i = 1, \ldots, n)$$

Basically, function *abstract* starts by flattening the input states by applying (zero or one steps of) rule replace, followed by (zero or more steps of) rule flatten. Function *abs* is defined inductively on the structure of flattened states (according to Lemma 1):

$$
\begin{array}{l}
abs(\mathcal{S}, \langle x, [\,]\rangle) = \mathcal{S} \\
abs(\mathcal{S}, \langle c(\overline{t_n}), [\,]\rangle) = abstract(\mathcal{S}, \mathcal{S}') \\
\quad \text{if } \overline{t'_m} \text{ are the maximal operation-rooted subterms of } c(\overline{t_n}) \text{ and } \mathcal{S}' = \{\overline{\langle t'_m, [\,]\rangle}\} \\
abs(\mathcal{S}, \langle f(\overline{t_n}), S\rangle) = \\
\quad \begin{cases}
\mathcal{S} \cup \{\langle f(\overline{t_n}), S\rangle\} & \text{if } \nexists \langle f(\overline{t'_n}), S'\rangle \in \mathcal{S} \\
\mathcal{S} & \text{else if } \langle f(\overline{t_n}), S\rangle \text{ is } \mathcal{S}\text{-closed} \\
abstract(\mathcal{S}^*, \mathcal{S}'') & \text{otherwise, where } \langle f(\overline{t'_n}), S'\rangle \in \mathcal{S}, \\
 & msg(\langle f(\overline{t'_n}), S'\rangle, \langle f(\overline{t_n}), S\rangle) = (\langle f(\overline{t''_n}), S'\rangle, \mathcal{S}''), \\
 & \text{and } \mathcal{S}^* = (\mathcal{S} \setminus \{\langle f(\overline{t'_n}), S'\rangle\}) \cup \{\langle f(\overline{t''_n}), S'\rangle\}
\end{cases}
\end{array}
$$

Informally speaking, function *abs* determines the corresponding action depending on the first component of the new state. If it is a variable, we discard the state. If it is constructor-rooted, we try to (recursively) add the maximal operation-rooted subterms. If it is a function call, then we have three possibilities:

– If there is no call to the same function in the current set, the new state is added to the current set of states.
– If there is a call to the same function in the current set, but the new call is *closed* w.r.t. this set, it is discarded.
– Otherwise, we generalize the new state and the existing state with the same outermost function—which is trivially unique by definition of *abstract*—and, then, we try to (recursively) add the states computed by function *msg*.

The notion of closedness is easily extended from expressions to states: a state $\langle t, S\rangle$ is *closed* w.r.t. a set of states $\mathcal{S}$ iff $S[t]$ is $T$-closed (according to Def. 1), with $T = \{S'[t'] \mid \langle t', S'\rangle \in \mathcal{S}\}$. Here, $S[t]$ denotes the term represented by $\langle t, S\rangle$, i.e., inner calls are moved back to their positions in the outer calls of the stack. For instance, given the state $\langle t, S\rangle = \langle \mathtt{y}, [(\mathtt{len}\ \mathtt{x_2},\ \mathtt{x_2}), (\mathtt{fst}\ (\mathtt{x_1},\ \mathtt{snd}\ \mathtt{z}),\ \mathtt{x_1})]\rangle$, we have $S[t] = \mathtt{fst}\ (\mathtt{len}\ \mathtt{y},\ \mathtt{snd}\ \mathtt{z})$.

The operator *msg* on states is defined as follows. First, we recall the standard notion of *msg* on terms: a term $t$ is a *generalization* of terms $t_1$ and $t_2$ if both $t_1$ and $t_2$ are instances of $t$; furthermore, term $t$ is the *msg* of $t_1$ and $t_2$ if $t$ is a generalization of $t_1$ and $t_2$ and, for any other generalization $t'$ of $t_1$ and $t_2$, $t$ is an instance of $t'$. Now, the *msg* of two states is defined by

$$msg(\langle t_1, S_1\rangle, \langle t_2, S_2\rangle) \ = \ (\langle t, S_1\rangle, calls(\sigma_1) \cup calls(\sigma_2) \cup calls(S_2))$$

where $msg(t_1, t_2) = t$, and $\sigma_1$ and $\sigma_2$ are the matching substitutions, i.e., $\sigma_1(t) = t_1$ and $\sigma_2(t) = t_2$. The auxiliary function *calls* returns a set of states of the form $\langle t, [\,]\rangle$ for each maximal operation-rooted term $t$ in (the range of) a substitution or in a stack. The abstraction operator is safe in the following sense:

**Lemma 2.** *Let $\mathcal{S}$ be a set of flattened states and $\mathcal{S}'$ a set of unfolded states (as returned by unfold). Then the states in $\mathcal{S} \cup \mathcal{S}'$ are closed w.r.t. abstract$(\mathcal{S}, \mathcal{S}')$.*

The above lemma is the key result to ensure the correctness of our approach. Indeed, it will allow us to prove that the generated slice is executable and that it contains all the functions which are needed to execute the slicing criterion.

*Example 3.* Let us consider again the program of Example 2. Given the slicing criterion "`foo [] y z`", the initial set of states is $\mathcal{S}_0 = \{\langle \texttt{foo [] y z}, [\,]\rangle\}$. Now, we show the sequence of iterations performed by the algorithm in Fig. 4:

$$\mathcal{S}'_0 = \{\langle \texttt{fst (len (app [] y), snd z)}, [\,]\rangle\}$$
$$\mathcal{S}_1 = \mathcal{S}_0 \cup \{\langle \texttt{app [] y}, [(\texttt{len x}_2\texttt{, x}_2), (\texttt{fst (x}_1\texttt{, snd z), x}_1)]\rangle\}$$

$$\mathcal{S}'_1 = \mathcal{S}'_0 \cup \{\langle \texttt{y}, [(\texttt{len x}_2\texttt{, x}_2), (\texttt{fst (x}_1\texttt{, snd z), x}_1)]\rangle\}$$
$$\mathcal{S}_2 = \mathcal{S}_1 \cup \{\langle \texttt{len y}, [(\texttt{fst (x}_1\texttt{, snd z), x}_1)]\rangle\}$$

$$\mathcal{S}'_2 = \mathcal{S}'_1 \cup \{\langle \texttt{Z}, [(\texttt{fst (x}_1\texttt{,snd z), x}_1)]\rangle, \langle \texttt{Succ (len ys)}, [(\texttt{fst (x}_1\texttt{,snd z), x}_1)]\rangle\}$$
$$\mathcal{S}_3 = \mathcal{S}_2 \cup \{\langle \texttt{fst (x}_3\texttt{, snd z)}, [\,]\rangle\}$$

$$\mathcal{S}'_3 = \mathcal{S}'_2 \cup \{\langle \texttt{x}_3, [\,]\rangle\} \quad \text{and} \quad \mathcal{S}_4 = \mathcal{S}_3$$

where $\mathcal{S}'_i = \mathit{unfold}(\mathcal{S}_i, \mathcal{R})$ and $\mathcal{S}_{i+1} = \mathit{abstract}(\mathcal{S}_i, \mathcal{S}'_i)$, for $i = 0, \ldots, 3$. Therefore, the algorithm returns the following set of states:

$$\begin{array}{rl}\mathcal{S} = \{ & \langle \texttt{foo [] y z}, [\,]\rangle,\ \langle \texttt{app [] y}, [(\texttt{len x}_2\texttt{, x}_2), (\texttt{fst (x}_1\texttt{, snd z), x}_1)]\rangle, \\ & \langle \texttt{len y}, [(\texttt{fst (x}_1\texttt{, snd z), x}_1)]\rangle,\ \langle \texttt{fst (x}_3\texttt{, snd z)}, [\,]\rangle \qquad \}\end{array}$$

The total correctness of the algorithm in Fig. 4 is stated in the following theorem:

**Theorem 1.** *Given a flat program $\mathcal{R}$ and an initial term $t$, the algorithm in Fig. 4 terminates computing a set of states $\mathcal{S}$ such that $\langle t, [\,]\rangle$ is $\mathcal{S}$-closed.*

### 3.3 Extraction of the Slice

The final step of the process consists in the construction of the residual program, i.e., the program slice. Let us recall that it must be a fragment of the original program—thus no instantiation of variables is allowed—and produce the same outputs for the slicing criterion as the original program. Therefore, we can only remove certain parts of the program: case alternatives which are not reachable from the slicing criterion as well as (inner) function calls which are not needed to evaluate the slicing criterion. While case alternatives can simply be discarded, inner calls which are not needed are replaced by a distinguished symbol $\top$ (in practice, any constant value may play the role of $\top$ since the evaluation of such a subterm will not be required). The interest in producing *executable* slices comes from the fact that it facilitates program reuse and, more importantly, it allows us to apply a number of existing techniques to the computed slice (e.g., debugging, program analysis, verification, program transformation, etc).

Let us assume that, given a program $\mathcal{R}$ and a term $t$, the algorithm of Fig. 4 returns the set of states $\mathcal{S}$. In principle, we could construct a residual program following the standard narrowing-driven specialization method as follows: for each state $\langle t, S\rangle \in \mathcal{S}$, we produce a residual rule $S[t] = S[t']$ (we ignore here the renaming of expressions), where $\langle t, S\rangle \Longrightarrow_{\mathsf{fun}} \langle t'', S\rangle \Longrightarrow^*_{\mathsf{select/guess}} \langle t', S\rangle \not\Longrightarrow_{\mathsf{select/guess}}$. The residual program constructed in this way would be correct in the sense that any computation for $t$ in $\mathcal{R}$ could also be performed in the residual program. In general, however, the residual program would not be a fragment of the original one since the left-hand sides of residual rules may contain non-variable arguments (even nested function calls). Therefore, we proceed as follows:

- Firstly, residual rules are constructed only for the first components of the states in $\mathcal{S}$. This is safe in our approach because the "context" (recorded in the stack) is not used when performing a complete one-step evaluation; moreover, if the evaluation of some call in the stack is required, it will appear in the first component of some other state.
- Secondly, we slightly change the rules of Fig. 3 so that only some case alternatives can be removed (if they are not reachable) but the case structure remains.
- Finally, in order to avoid the instantiation of variables, the new rules maintain separately the program expression being reduced and the bindings for the program variables.

The slice built in this way is appropriate *when the extended operational semantics of Fig. 3 is considered.* Unfortunately, this is no longer true under the standard operational semantics when the program contains *non-terminating* functions. Consider, for instance, the following program rules:

```
g x = g x              f x = case x of { [] → [] }
```

Given the initial term "f (g x)", whose flattening is $\langle \mathtt{g\ x}, [(\mathtt{f\ y,\ y})]\rangle$, the algorithm of Fig. 4 returns the final set $\{\langle \mathtt{g\ x}, [(\mathtt{f\ y,\ y})]\rangle\}$ since the evaluation of "f y" is not required. Therefore, the computed slice would be the rule $\mathtt{g\ x} = \mathtt{g\ x}$. Under the extended semantics, the initial state $\langle \mathtt{f\ (g\ x)},\ [\,]\rangle$ performs exactly the same steps in both the original program and the computed slice. However, under the standard semantics (where outer function calls are not delayed), the initial term "f (g x)" is reduced to "case (g x) of $\{[] \to []\}$" (and then enters into an infinite loop) which is not possible in the slice. In order to have a complete equivalence w.r.t. the standard semantics, we also need to add residual definitions for those functions in the stacks which are not closed w.r.t. the set of first components of the states in $\mathcal{S}$. The following auxiliary function returns all the relevant terms:

$$\mathit{residual\_calls}(\mathcal{S}) = T_{\mathcal{S}} \cup \{t' \mid \langle t, S\rangle \in \mathcal{S},\ t' \in \mathit{calls}(S),\ \textsf{and } t' \textsf{ is not } T_{\mathcal{S}}\textsf{-closed}\}$$

where $\mathcal{S}$ is the set of states returned by the algorithm of Fig. 4 and

$$T_{\mathcal{S}} = \{t \mid \langle t, S\rangle \in \mathcal{S}\}$$

The program slice is then computed by using the following function:

$$\begin{array}{l} \mathit{build\_slice}(T) = \textsf{if } T = \{\,\} \textsf{ then } \{\,\} \textsf{ else } \{f(\overline{x_n}) = e'\} \cup \mathit{build\_slice}(T') \\ \qquad\qquad\qquad\qquad \textsf{where } T = \{f(\overline{t_n})\} \cup T',\ f(\overline{x_n}) = e \in \mathcal{R},\ \overline{y_n} \textsf{ are fresh,} \\ \qquad\qquad\qquad\qquad\qquad\quad \rho = \{\overline{y_n \mapsto t_n}\},\ \textsf{and } [\![e]\!]\rho \longrightarrow^* e' \not\longrightarrow \end{array}$$

| Rule | *Expr* | $\longrightarrow$ *Expr* |
|---|---|---|
| var | $[\![x]\!]\rho$ | $\longrightarrow x$ |
| cons | $[\![c(\overline{t_n})]\!]\rho$ | $\longrightarrow c([\![t_1]\!]\rho, \ldots, [\![t_n]\!]\rho)$ |
| select | $[\![(f)case\ x\ of\ \{\overline{p_k \to e_k}\}]\!]\rho$ | $\longrightarrow (f)case\ x\ of\ \{\overline{p_i \to [\![e_i]\!]\rho'}\}$ |
| guess | $[\![(f)case\ x\ of\ \{\overline{p_k \to e_k}\}]\!]\rho$ | $\longrightarrow (f)case\ x\ of\ \{\overline{p_k \to [\![e_k]\!]\rho_k}\}$ |
| fun | $[\![f(\overline{t_n})]\!]\rho$ | $\longrightarrow f([\![t_1]\!]\rho, \ldots, [\![t_n]\!]\rho)$ |
| remove | $[\![f(\overline{t_n})]\!]\rho$ | $\longrightarrow \top$ |

where in select: $\rho(x) = c(\overline{t_n}),\ p_i = c(\overline{x_n}),\ \rho' = \{\overline{x_n \mapsto t_n}\} \circ \rho,$ and $i \in \{1, \ldots, k\}$
guess: $\rho(x) \in \mathcal{X},\ \rho_i = \{x \mapsto p_i\} \circ \rho,$ and $i \in \{1, \ldots, k\}$
fun: $\rho(f(\overline{t_n}))$ is closed w.r.t. $residual\_calls(\mathcal{S})$
remove: $\rho(f(\overline{t_n}))$ is not closed w.r.t. $residual\_calls(\mathcal{S})$

**Fig. 5.** Simplified Unfolding Rules

In order to extract the program slice, function *build_slice* is called with the result of function *residual_calls*, i.e., the initial call is as follows:

$$build\_slice(residual\_calls(\mathcal{S}))$$

where the set $\mathcal{S}$ is the output of the algorithm in Fig. 4. The new calculus which is used to construct the rules of the slice is depicted in Fig. 5. First, note that the symbols "$[\![$" and "$]\!]$" in an expression like $[\![e]\!]\rho$ are purely syntactical, i.e., they are only used to mark subexpressions where the inference rules may be applied. The substitution $\rho$ is used to store the bindings for the program variables. Let us briefly explain the rules of the new calculus. Rule var simply returns a variable unchanged. Rule cons applies to constructor-rooted terms; it leaves the constructor symbol and, then, it is (recursively) applied to inspect the arguments. Rules select and guess proceed similarly to their counterpart in Fig. 3 but leave the case structure; the substitution $\rho$ is used to check the current value of the case argument. We only deal with *variable* case arguments since the considered expression is the right-hand side of some program rule (see Fig. 1). Note that rule guess is now deterministic (and, thus, the entire calculus). Finally, rules fun and remove are used to reduce function calls: when the function call is closed w.r.t. $residual\_calls(\mathcal{S})$, we proceed as in rule cons; otherwise, we return $\top$ (which means that the evaluation of this function call is not needed).

The following example illustrates the computation of a program slice.

*Example 4.* Let us consider the set of states computed in Example 3. From this set, function *residual_calls* returns the set of terms:

{`foo [] y z`, `app [] y`, `len y`, `fst (x, snd z)`}

Now, we construct a residual rule for each term of the set. For "`foo [] y z`", the residual rule is: "`foo x y z = fst (len (app x y),` $\top$`)`", since the following derivation is possible (with $\rho = \{x \mapsto [\,]\}$):

$$
\begin{aligned}
[\![\texttt{fst (len (app x y), snd z)}]\!]\rho &\longrightarrow^*_{\mathsf{fun}} \texttt{fst (len (app } [\![\texttt{x}]\!]\rho\ [\![\texttt{y}]\!]\rho), [\![\texttt{snd z}]\!]\rho) \\
&\longrightarrow^*_{\mathsf{var}} \texttt{fst (len (app x y)}, [\![\texttt{snd z}]\!]\rho) \\
&\longrightarrow_{\mathsf{remove}} \texttt{fst (len (app x y)}, \top)
\end{aligned}
$$

By constructing a residual rule associated to each of the remaining terms, the computed slice coincides with the one which is shown in Example 2.

The computed slice is executable and contains all the functions which are needed to evaluate the slicing criterion. This property is inherited by the correctness of the underlying partial evaluation process.

**Theorem 2.** *Let $\mathcal{R}$ be a flat program and $t$ a term. Let $\mathcal{S}$ be a set of states computed by the algorithm in Fig. 4 from $\mathcal{R}$ and $t$. Then $t$ computes the same values in $\mathcal{R}$ and in $build\_slice(residual\_calls(\mathcal{S}))$.*

This section has shown the definition of a practical forward slicing technique based on a partial evaluation scheme. In the next section, we discuss several possibilities to design a backward slicing method following a similar style.

## 4 Backward Slicing

In this section, we informally explain how the scheme presented so far could be extended in order to perform (static/dynamic) backward slicing. In principle, backward slicing can be seen from two different perspectives (in either case, we consider that the program contains a distinguished function, *main*, which is used to start the execution of the program):

- Type I: A naive approach to backward slicing implies extracting those statements of the original program that may reach the slicing criterion, when the program is executed starting at function *main*.
- Type II: A different notion of backward slicing has been introduced in [24] to perform program slicing of functional programs. In this work, the slicing criterion is some *part* of the output of function *main* (described by means of a projection). Then the method extracts those program statements which are needed to compute the desired fragment of the output.

A naive approach to Type I backward slicing can easily be defined from the technique presented so far. Currently, the algorithm in Fig. 4 only computes the set of reachable function calls, starting at the slicing criterion. Now, we should start the algorithm with a call to function *main*. However, in contrast to the approach of the previous section, we should explicitly compute the graph of dependences, i.e., it is not sufficient to have the set of reachable nodes but we also need the relationships among them. By traversing this graph backwards (from the slicing criterion), we could easily select the desired nodes and, then, construct its associated slice (by using the same program extraction method of Sect. 3.3).

Type II backward slicing is more useful but also more complex. Nevertheless, we could still adapt the previous developments in order to cope with this form of slicing. Functional logic languages naturally support some form of constraint solving; indeed, they accept the evaluation of either function calls $f(\overline{t_n})$ and equational constraints $f(\overline{t_n}) =:= t$. In this context, the slicing criterion can be defined by providing an equational constraint $main(\overline{t_n}) =:= t$, where $t$ is a term which contains free variables for those parts of the output which are not relevant for the slice, and some special values for those parts of the output whose computation is required. Then the algorithm in Fig. 4 could be adapted to work with equational constraints and only evaluate those function calls which are needed to produce the outputs determined by $t$. The resulting method would share many similarities with the technique for backward slicing of [24].

These approaches are subject of ongoing work. It would be also interesting to relate backward slicing with recent approaches to function inversion [1, 26].

## 5 Implementation

In order to check the practicality of the ideas presented so far, a prototype implementation of the program slicing tool has been developed.[4] In particular, it has been implemented by adapting an existing partial evaluator for Curry programs [3]. The resulting tool covers not only the flat programs of Sect. 2, but source Curry programs (which are automatically translated to the flat syntax). We accept higher-order functions, overlapping left-hand sides, several predefined functions, etc; all these features were also available in the underlying partial evaluator. It required a small implementation effort, i.e., only the underlying meta-interpreter needed significant changes.

Our slicing tool is able to compute the slice of Example 2, thus it is strictly more powerful than naive approaches based on graphs of functional dependences. Preliminary results are quite encouraging. In fact, in contrast to the original partial evaluator, it can deal with larger programs efficiently; this is mainly due to the monovariant/monogenetic nature of the basic algorithm, which simplifies the computation of a closed set of terms.

## 6 Conclusions and Related Work

This work presents the first approach to forward slicing of multi-paradigm (functional logic) programs. Our developments rely on adapting an online partial evaluation scheme for such programs. Thus, the implementation of the resulting slicing tool can easily be undertaken by adapting existing partial evaluation tools. Moreover, our approach helps to clarify the relation between program slicing and partial evaluation in a functional logic context. The application of our developments to pure (lazy) functional programs would be straightforward, since the considered language is a conservative extension of a pure lazy functional language and the (online) partial evaluation techniques are similar (e.g., positive supercompilation [28]).

[4] It is publicly available from `http://www.dsic.upv.es/~gvidal`.

As mentioned in the introduction, we found very few approaches to program slicing in the context of declarative programming. Some exceptions are [13, 21, 24, 25, 29]). Among them, the closest to our work is [24], the only of them which considers a functional syntax for programs. In contrast to our approach, [24] defines a *backward* slicing technique. As in our case, some of his developments are inspired by previous techniques coming from partial evaluation (like, e.g., [18, 23]). In the context of imperative programs, [8] has shown how forward slicing can be used to carry out binding-times analyses for imperative programs. Our approach can be seen as complementary: we show how to use (online) partial evaluation to perform forward slicing in a declarative context.

Future work includes the definition of appropriate algorithms to perform backward slicing (as discussed in Sect. 4). It would be also interesting to investigate alternative approaches to program slicing based on abstract interpretation (e.g., by approximating the operational semantics presented in Sect. 3.2). A different line of research involves the definition of a forward slicing technique for logic programs by exploiting the similarities between narrowing-driven specialization and conjunctive partial deduction [9]. In this context, precision could be improved by considering refined frameworks for partial deduction like, e.g., *abstract* partial deduction [20]. This topic is subject of ongoing work.

**Acknowledgements**

We gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2002 for many useful comments and suggestions.

## References

1. S.M. Abramov and R. Glück. The Universal Resolving Algorithm: Inverse Computation in a Functional Language. In *Mathematics of Program Construction. Proceedings*, pages 187–212. Springer LNCS 1837, 2000.
2. E. Albert, M. Hanus, F. Huch, J. Olvier, and G. Vidal. Operational Semantics for Functional Logic Languages. In *Proc. of the Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP'02)*, volume 76 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.
3. E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
4. E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.
5. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.
6. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
7. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
8. M. Das, T. Reps, and P. Van Hentenryck. Semantic Foundations of Binding-Time Analysis for Imperative Programs. In *Proc. of the Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'95)*, pages 100–110, 1995.

9. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
10. J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
11. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98. ACM, New York, 1993.
12. R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 137–160. Springer LNCS 1110, 1996.
13. V. Gouranton. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proc. of SAS'98*, pages 115–133, 1998.
14. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, New York, 1997.
15. M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.2: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000.
16. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
17. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.
18. J. Hughes. Backwards Analysis of Functional Programs. In *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 187–208. North-Holland, Amsterdam, 1988.
19. D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimization. In *Proc. of the 8th Symp. on the Principles of Programming Languages (POPL'81), SIGPLAN Notices*, pages 207–218, 1981.
20. M. Leuschel. Program Specialization and Abstract Interpretation Reconciled. In *Proc. of the Joint Int'l Conf. and Symp. on Logic Programming (JICSLP'98)*, pages 220–234. MIT Press, 1998.
21. M. Leuschel and M.H. Sørensen. Redundant Argument Filtering of Logic Programs. In *Proc. of LOPSTR'96*, pages 83–103. LNCS 1207 83–103, 1996.
22. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
23. T. Mogensen. Separating Binding-Times in Language Specifications. In *Proc. of 4th Int'l Conf. on Functional Programming and Computer Architecture (FPCA'89)*, pages 12–25. ACM, New York, 1989.
24. T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 409–429. Springer LNCS 1110, 1996.
25. S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Proc. of SAS'96*, pages 317–331. Springer LNCS 1145, 1996.
26. J.P. Secher and M.H. Sørensen. From Checking to Inference via Driving and Dag Grammars. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 41–51. ACM, New York, 2002.
27. P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

28. M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
29. G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.
30. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
31. V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
32. G. Vidal. Forward Slicing by Partial Evaluation. Technical report, DSIC, Technical University of Valencia, 2003. Available from `http://www.dsic.upv.es/~gvidal`.
33. M.D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.
34. M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.