TLP **7** (1&2): 215–247, 2007. © 2007 Cambridge University Press
doi:10.1017/S1471068406002870 Printed in the United Kingdom

*Forward slicing of functional logic programs by partial evaluation**

JOSEP SILVA and GERMÁN VIDAL

DSIC, Technical University of Valencia,
Camino de Vera s/n, E-46022 Valencia, Spain
(*e-mail:* {jsilva,gvidal}@dsic.upv.es)

submitted 5 January 2004; revised 20 February 2005; accepted 5 January 2006

Abstract

Program slicing has been mainly studied in the context of imperative languages, where it has been applied to a wide variety of software engineering tasks, like program understanding, maintenance, debugging, testing, code reuse, etc. This work introduces the first forward slicing technique for declarative multi-paradigm programs which integrate features from functional and logic programming. Basically, given a program and a *slicing criterion* (a function call in our setting), the computed forward slice contains those parts of the original program which are *reachable* from the slicing criterion. Our approach to program slicing is based on an extension of (online) partial evaluation. Therefore, it provides a simple way to develop program slicing tools from existing partial evaluators and helps to clarify the relation between both methodologies. A slicing tool for the multi-paradigm language Curry, which demonstrates the usefulness of our approach, has been implemented in Curry itself.

KEYWORDS: forward slicing, partial evaluation, functional logic programming

1 Introduction

Essentially, program slicing is a method for decomposing programs by analyzing their data and control flow. It was first proposed as a debugging tool to allow a better understanding of the portion of code which revealed an error. Since this concept was originally introduced by Weiser (1979; 1984) – in the context of imperative programs – it has been successfully applied to a wide variety of software engineering tasks (e.g., program understanding, maintenance, debugging, merging, testing, code reuse). Surprisingly, there are very few approaches to program slicing in the context of *declarative* programming (see Section 8).

* A preliminary short version of this paper appeared in the Proceedings of the *12th International Workshop on Logic Based Program Synthesis and Transformation* (LOPSTR 2002).
This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2004-00231 and TIN2005-09207-C03-02, and by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

```
(1) read(n);                (1) read(n);                (1)
(2) i := 1;                 (2) i := 1;                 (2)
(3) sum := 0;               (3)                         (3) sum := 0;
(4) prod := 1;              (4) prod := 1;              (4)
(5) while i <= n do         (5) while i <= n do         (5)
(6)    sum := sum + i;      (6)                         (6)    sum := sum + i;
(7)    prod := prod * i;    (7)    prod := prod * i;    (7)
(8)    i := i + 1;          (8)    i := i + 1;          (8)
(9) write(sum);             (9)                         (9) write(sum);
(10)write(prod);            (10)write(prod);            (10)
```

(a) (b) (c)

Fig. 1. Forward and backward slicing – an example.

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, often given by a pair `(line number, variable)`, referred to as a *slicing criterion*. Program slices are usually computed from a *program dependence graph* (Ferrante *et al.* 1987; Kuck *et al.* 1981) that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards and forwards – from the slicing criterion – giving rise to so-called *backward* and *forward* slicing, respectively.

Essentially, a backward slice consists of the parts of the program that (potentially) affect the values computed at the slicing criterion. In contrast, a forward slice consists of the statements which are dependent on the slicing criterion, a statement being *dependent* on the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion or if the values computed at the slicing criterion determine if the statement under consideration is executed (Tip 1995). Consider, e.g., the example (Tip 1995) depicted in Figure 1 (a) for computing the sum and the product of the sequence of numbers `1,2,...,n`. Figure 1 (b) shows a backward slice of the program w.r.t. the slicing criterion `(10,prod)` while Figure 1 (c) shows a forward slice w.r.t. the slicing criterion `(3,sum)`.

Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. *Quasi* static slicing was the first attempt to define a hybrid method ranging between static and dynamic slicing (Venkatesh 1991). It becomes useful when only the value of some parameters is known. This notion is closely related to partial evaluation (Jones *et al.* 1993), a well-known technique to specialize programs w.r.t. part of their input data. For instance, quasi static slicing has been applied to program understanding by Harman *et al.* 1995; similarly, Blazy and Facon (1998) use partial evaluation for the same purpose.

All approaches to slicing mentioned so far are *syntax preserving*, i.e., they are mainly obtained from the original program by statement deletion. In contrast, *amorphous* slicing (Harman and Danicic 1997) exploits different program transformations in order to simplify the program while preserving its semantics w.r.t. the slicing criterion. From this perspective, partial evaluation could straightforwardly be seen as an amorphous slicing technique. More detailed information on program slicing can be found in the surveys of Harman and Hierons (2001) and Tip (1995).

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

The aim of this work is the definition of a forward slicing technique for a multi-paradigm declarative language which integrates features from functional and logic programming, like, e.g., Curry (Hanus 2003) or Toy (López-Fraguas and Sánchez-Hernández 1999). Similar to Reps and Turnidge (1996), where a first-order functional language is considered, given a program p and a projection function π, backward slicing should extract a program that behaves like $\pi(p)$ (e.g., by symbolically pushing π backwards through the body of p). For instance, it can be used to extract a program slice for computing the number of lines in a string from a more general program that returns a tuple with both the number of lines and the number of characters of the string; this is the example that illustrates the backward slicing technique of Reps and Turnidge (1996). Such a slicing technique is considered *backward* because the algorithm proceeds from (part of) the result backwards to the initial function call, i.e., in the inverse direction of the standard operational semantics. In contrast, here we consider the definition of a forward slicing technique that, given a program and a function call, extracts a program containing all the statements which are *reachable* from the slicing criterion. Our slicing technique is considered *forward* because it proceeds from a given function call to its result, i.e., we follow the control flow of the standard operational semantics.

Furthermore, rather than defining a new technique from scratch, we exploit the similarities between slicing and *partial evaluation* (Jones *et al.* 1993). Since a partial evaluator for the considered language already exists, our approach provides a simple way to develop a program slicing tool. The main purpose of partial evaluation is to specialize a program w.r.t. part of its input data and, hence, it is also known as *program specialization*. The partially evaluated program will be (hopefully) executed more 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 function call (possibly containing partial data structures by means of free variables), the partial evaluator builds a *finite* representation – generally a graph – of the possible executions of the initial call and, then, systematically extracts a *residual* program – the partially evaluated program – from this graph.

The essence of our approach can be summarized as follows. First, we consider that, in our functional logic context, a function call – possibly containing free variables – may also play the role of *slicing criterion*. Since such a call may have an infinite computation space, a primary task of both slicing and partial evaluation is the construction of a finite representation of its possible program executions. Here, the same algorithm which is used in partial evaluation can be applied for computing this finite representation, which will be later used to identify the program statements that are reachable from the slicing criterion. Then, we only need to replace the construction of a residual program in partial evaluation by a simpler post-processing stage that extracts an executable program which includes the reachable program statements.

While partial evaluation usually achieves its effects by compressing paths in the graph and by renaming expressions to remove unnecessary function symbols, slicing should preserve the structure of the original program (here, we do not consider amorphous slicing): statements can be – totally or partially – deleted but no new

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

statements can be introduced. In order to further clarify the relation between partial evaluation and slicing, let us recall the following classification of partial evaluators introduced by Glück and Sørensen (1996). According to this classification, a partial evaluator is

- *Monovariant*: if each function of the original program gives rise to (at most) one residual function;
- *Polyvariant*: if each function of the original program may give rise to one or more residual functions;
- *Monogenetic*: if each residual function stems from one function of the original program;
- *Polygenetic*: if each residual function may stem from one or more functions of the original program.

The main contribution of this work is to demonstrate that a forward slicing technique for functional logic programs can be obtained by slightly extending a *monovariant* and *monogenetic* partial evaluation scheme. Unfortunately, this kind of monovariant/monogenetic partial evaluation could be rather imprecise, thus resulting in unnecessarily large residual programs (i.e., slices). To overcome this drawback, we consider the definition of 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.

The main contributions of this work can be summarized as follows:

- We define the first forward slicing technique for functional logic programs. Furthermore, the application of our developments to (first-order) lazy functional programs would be straightforward, since either the syntax and the underlying (online) partial evaluators (e.g., positive supercompilation (Sørensen *et al.* 1996)) share many similarities.
- We do not need to consider separately static and dynamic slicing, since the underlying partial evaluation scheme naturally accepts partial input data.
- Our method is defined in terms of an existing partial evaluation scheme and, thus, it is easy to implement by adapting current partial evaluators.
- Finally, our approach helps to clarify the relation between forward slicing and (online) partial evaluation.

This paper is organized as follows. In the next section we recall some foundations for understanding the subsequent developments. Section 3 introduces a notion of forward slicing in the context of functional logic programming. We then recall, in Section 4, the narrowing-driven approach to partial evaluation. Section 5 defines an algorithm for computing program dependences by partial evaluation, while Section 6 uses these dependences to extract program slices. Section 7 presents a prototype implementation of the program slicing tool and show some selected experiments. Several related works are discussed in Section 8 before we conclude in Section 9. Detailed proofs of technical results can be found in Silva and Vidal (2006).

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

2 Foundations

We recall in this section some basic notions of term rewriting (Baader and Nipkow 1998; Klop 1992; Terese 2003) and functional logic programming (Hanus 1994).

2.1 Preliminaries

Throughout this paper, we consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathscr{C}$ of *constructors* and a set $\mathscr{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathscr{C}$ and $f/n \in \mathscr{F}$ for n-ary constructor and operation symbols, respectively. There is at least one sort *Bool* containing the constructors `True` and `False`. The set of *terms* and *constructor terms* with *variables* (e.g., x, y, z) from $\mathscr{X}$ are denoted by $\mathscr{T}(\mathscr{C} \cup \mathscr{F}, \mathscr{X})$ and $\mathscr{T}(\mathscr{C}, \mathscr{X})$, respectively. A term is *linear* if it does not contain multiple occurrences of one variable. The set of variables occurring in a term t is denoted by $\mathscr{V}ar(t)$. A term t is *ground* if $\mathscr{V}ar(t) = \emptyset$.

A *pattern* is a term of the form $f(d_1, \ldots, d_n)$ where $f/n \in \mathscr{F}$ and $d_1, \ldots, d_n \in \mathscr{T}(\mathscr{C}, \mathscr{X})$. A term is *operation-rooted* (constructor-rooted) if it has an operation (constructor) symbol at the root. A *position* p in a term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. We denote a *substitution* σ by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. A substitution σ is *constructor*, if $\sigma(x)$ is a constructor term for all x. The identity substitution is denoted by id. A substitution θ is more general than σ, in symbols $\theta \leqslant \sigma$, iff there exists a substitution γ such that $\gamma \circ \theta = \sigma$ ("$\circ$" denotes the composition operator). Term t' is a (constructor) *instance* of term t if there is a (constructor) substitution σ with $t' = \sigma(t)$.

A set of rewrite rules (or oriented equations) $l = r$ such that $l \notin \mathscr{X}$, and $\mathscr{V}ar(r) \subseteq \mathscr{V}ar(l)$ is called a *term rewriting system* (TRS). Terms l and r are called the *left-hand side* and the *right-hand side* of the rule, respectively. A TRS $\mathscr{R}$ is *left-linear* if l is linear for all $l = r \in \mathscr{R}$. A TRS is *constructor-based* if each left-hand side l is a pattern. In the following, a functional logic *program* is a left-linear constructor-based TRS. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = (l = r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. The instantiated left-hand side $\sigma(l)$ of a rule $l = r$ is called a *redex* (*red*ucible *ex*pression). Given a relation $\to$, we denote by $\to^*$ its transitive and reflexive closure.

Example 1
Consider the following TRS that defines the addition on natural numbers represented by terms built from `Zero` and `Succ`:[1]

$$\begin{array}{rcll} \texttt{Zero} + \texttt{y} & = & \texttt{y} & (\mathrm{R}_1) \\ \texttt{Succ(x)} + \texttt{y} & = & \texttt{Succ(x + y)} & (\mathrm{R}_2) \end{array}$$

[1] In the examples, we write constructor symbols starting with upper case (except for the list constructors, "`[]`" and "`:`", which are a shorthand for `Nil` and `Cons`, respectively).

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Given the term $\texttt{Succ(Zero)} + \texttt{Succ(Zero)}$, we have the following sequence of rewrite steps:

$$\begin{array}{lll} \texttt{Succ(Zero)} + \texttt{Succ(Zero)} & \rightarrow_{\Lambda,\texttt{R}_2} & \texttt{Succ(Zero} + \texttt{Succ(Zero)} \\ & \rightarrow_{1,\texttt{R}_1} & \texttt{Succ(Succ(Zero))} \end{array}$$

2.2 Narrowing

Functional *logic* programs mainly differ from purely functional programs in that function calls may contain *free* variables. In order to evaluate terms containing free variables, *narrowing* non-deterministically instantiates these variables so that a rewrite step is possible. Formally, $t \leadsto_{(p,R,\sigma)} t'$ is a *narrowing step* if p is a non-variable position of t and $\sigma(t) \rightarrow_{p,R} t'$. We often write $t \leadsto_\sigma t'$ when the position and the rule are clear from the context. We denote by $t_0 \leadsto_\sigma^n t_n$ a sequence of n narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$), usually restricted to the variables of t_0. Due to the presence of free variables, a term may be reduced to different values after instantiating these variables to different terms. Given a narrowing derivation $t_0 \leadsto_\sigma^* t_n$, we say that t_n is a computed *value* and σ is a computed *answer* for t_0.

Example 2
Consider again the definition of function "+" in Example 1. Given the term $\texttt{x} + \texttt{Succ(Zero)}$, narrowing non-deterministically performs the following derivations:

$$\begin{array}{lll} \texttt{x} + \texttt{Succ(Zero)} & \leadsto_{\Lambda,\texttt{R}_1,\{\texttt{x}\mapsto\texttt{Zero}\}} & \texttt{Succ(Zero)} \\ \texttt{x} + \texttt{Succ(Zero)} & \leadsto_{\Lambda,\texttt{R}_2,\{\texttt{x}\mapsto\texttt{Succ(y}_1\texttt{)}\}} & \texttt{Succ(y}_1 + \texttt{Succ(Zero))} \\ & \leadsto_{1,\texttt{R}_1,\{\texttt{y}_1\mapsto\texttt{Zero}\}} & \texttt{Succ(Succ(Zero))} \\ \texttt{x} + \texttt{Succ(Zero)} & \leadsto_{\Lambda,\texttt{R}_2,\{\texttt{x}\mapsto\texttt{Succ(y}_1\texttt{)}\}} & \texttt{Succ(y}_1 + \texttt{Succ(Zero))} \\ & \leadsto_{1,\texttt{R}_2,\{\texttt{y}_1\mapsto\texttt{Succ(y}_2\texttt{)}\}} & \texttt{Succ(Succ(y}_2 + \texttt{Succ(Zero)))} \\ & \leadsto_{1\cdot 1,\texttt{R}_1,\{\texttt{y}_2\mapsto\texttt{Zero}\}} & \texttt{Succ(Succ(Succ(Zero)))} \\ \ldots & & \end{array}$$

Therefore, $\texttt{x} + \texttt{Succ(Zero)}$ non-deterministically computes the values

- $\texttt{Succ(Zero)}$ with answer $\{\texttt{x} \mapsto \texttt{Zero}\}$,
- $\texttt{Succ(Succ(Zero))}$ with answer $\{\texttt{x} \mapsto \texttt{Succ(Zero)}\}$,
- $\texttt{Succ(Succ(Succ(Zero)))}$ with answer $\{\texttt{x} \mapsto \texttt{Succ(Succ(Succ(Zero)))}\}$, etc.

As in logic programming, narrowing derivations can be represented by a (possibly infinite) finitely branching *tree*. Formally, given a program $\mathscr{R}$ and an operation-rooted term t, a *narrowing tree* for t in $\mathscr{R}$ is a tree satisfying the following conditions: (a) each node of the tree is a term, (b) the root node is t, (c) if s is a node of the tree then, for each narrowing step $s \leadsto_{p,R,\sigma} s'$, the node has a child s' and the corresponding arc is labeled with (p, R, σ), and (d) nodes which are constructor terms have no children.

In order to avoid unnecessary computations and to deal with infinite data structures, demand-driven generation of the search space has been advocated by a number *lazy* narrowing strategies (Giovannetti *et al.* 1991; Loogen *et al.* 1993;

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Moreno-Navarro and Rodríguez-Artalejo 1992). Due to its optimality properties w.r.t. the length of derivations and the number of computed solutions, *needed narrowing* (Antoy *et al.* 2000) is currently the best lazy narrowing strategy.

2.3 Needed narrowing

Needed narrowing (Antoy *et al.* 2000) is defined on *inductively sequential* TRSs (Antoy 1992), a subclass of left-linear constructor-based TRSs. Essentially, a TRS is *inductively sequential* when all its operations are defined by rewrite rules that, recursively, make on their arguments a case distinction analogous to a data type (or structural) induction. Inductive sequentiality is not a limiting condition for programming. In fact, the first-order components of many functional (logic) programs written in, e.g., Haskell, ML or Curry, are inductively sequential.

We say that $s \leadsto_{p,R,\sigma} t$ is a *needed narrowing step* iff $\sigma(s) \rightarrow_{p,R} t$ is a *needed rewrite* step in the sense of Huet and Lévy 1992, i.e., in every computation from $\sigma(s)$ to a normal form, either $\sigma(s)|_p$ or one of its *descendants* must be reduced. Here, we are interested in a particular needed narrowing strategy, denoted by λ in Antoy *et al.* (2000, Def. 13) which is based on the notion of a *definitional tree* (Antoy 1992), a hierarchical structure containing the rules of a function definition, which is used to guide the needed narrowing steps. This strategy is basically equivalent to *lazy narrowing* (Moreno-Navarro and Rodríguez-Artalejo 1992) where narrowing steps are applied to the outermost function, if possible, and inner functions are only narrowed if their evaluation is *demanded* by a constructor symbol in the left-hand side of some rule (i.e., a typical outermost strategy).

Example 3
Consider following rules which define the less-or-equal function on natural numbers:

```
  Zero ⩽ y       =  True
Succ(x) ⩽ Zero   =  False
Succ(x) ⩽ Succ(y) = x ⩽ y
```

In a term like $t_1 \leqslant t_2$, it is always necessary to evaluate t_1 to some *head normal form* (i.e., a variable or a constructor-rooted term) since all three rules defining "$\leqslant$" have a non-variable first argument. On the other hand, the evaluation of t_2 is only *needed* if t_1 is of the form `Succ(`t`)`. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor, here `Zero` or `Succ(x)`. Depending on this instantiation, either the first rule is applied or the second argument t_2 is evaluated.

2.4 Declarative multi-paradigm languages

Functional logic languages have recently evolved to so called declarative *multi-paradigm* languages like Curry (Hanus 2003), Toy (Hortalá-González and Ullán 2001) and Escher (Lloyd 1994). To make things concrete, we consider in this work the language Curry, a modern multi-paradigm language which integrates features from logic programming (partial data structures, built-in search), functional programming (higher-order functions, demand-driven evaluation) and concurrent programming

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

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

Fig. 2. Syntax of flat programs.

(concurrent evaluation of constraints with synchronization on logical variables). Curry follows a Haskell-like syntax, i.e., variables and function names start with lowercase letters and data constructors start with an uppercase letter. The application of function f to an argument e is denoted by juxtaposition, i.e., $(f\ e)$.

The basic operational semantics of Curry is based on a combination of needed narrowing and residuation (Hanus 1997). The *residuation* principle is based on the idea of delaying function calls until they are ready for a deterministic evaluation. Residuation preserves the deterministic nature of functions and naturally supports concurrent computations. The precise mechanism – narrowing or residuation – for each function is specified by *evaluation annotations*. The annotation of a function as *rigid* forces the delayed evaluation by rewriting, while functions annotated as *flexible* can be evaluated in a non-deterministic manner by narrowing.

In actual implementations, e.g., the PAKCS environment (Hanus *et al.* 2004) for Curry, 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 Hanus and Prehofer (1999) to express pattern-matching by case expressions. The complete flat representation is called FlatCurry (Hanus *et al.* 2004) and is used as an intermediate language during the compilation of source programs.

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 Section 7 covers many of these features. The syntax of flat programs is summarized in Figure 2, where $\overline{o_n}$ stands for the *sequence* of objects $o_1, \ldots, o_n$. We consider the following domains:

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

The only difference between *terms* and *expressions* is that the latter may contain case expressions. *Values* are terms in head normal form, i.e., variables or constructor-rooted terms. A program $\mathscr{R}$ consists of a sequence of function definitions; 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

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

variables, constructors, function calls, and case expressions for pattern-matching. The general form of a case expression is:[2]

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

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 $(f)case$'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 (Antoy *et al.* 2000) and driving (Turchin 1986). Note that our functional logic language mainly differs from typical (lazy) functional languages in the presence of flexible case expressions.

Example 4

Consider again the rules defining functions "+" (Example 1) and "$\leqslant$" (Example 3). These functions can be defined in the flat representation as follows:[3]

```
x + y = fcase x of {Zero   → y;
                    Succ n → Succ (n + y)}

x ⩽ y = fcase x of {Zero   → True;
                    Succ n → fcase y of {Zero   → False;
                                         Succ m → n ⩽ m}}
```

An automatic transformation from source (inductively sequential) programs to flat programs has been introduced by Hanus and Prehofer (1999). 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 Figure 2 is general enough for our purposes. We shall assume these restrictions on flat programs in the following.

The operational semantics of flat programs is shown in Figure 3. It is based on the LNT – for Lazy Narrowing with definitional Trees – calculus of Hanus and Prehofer (1999). The one-step transition relation $\Longrightarrow_\sigma$ is labeled with the substitution σ computed in the step. Let us briefly describe the LNT rules:

The select rule selects the appropriate branch of a case expression and continues with the evaluation of this branch. This rule implements *pattern matching*.

The guess rule applies when the argument of a *flexible* case expression is a variable. Then, this rule non-deterministically binds this variable to a pattern in a branch of the case expression. The step is labeled with the computed binding.

[2] We write $(f)case$ for either *fcase* or *case*.

[3] Although we consider in this work a first-order representation – the flat language – we use a curried notation in concrete examples (as in Curry).

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

(select)	$(f)case\ c(\overline{t_n})\ of\ \{\overline{p_m \to e_m}\}$	$\Longrightarrow_{id}\ \sigma(e_i)$ if $p_i = c(\overline{x_n})$, $c \in \mathscr{C}$, and $\sigma = \{\overline{x_n \mapsto t_n}\}$
(guess)	$fcase\ x\ of\ \{\overline{p_m \to e_m}\}$	$\Longrightarrow_{\sigma}\ \sigma(e_i)$ if $\sigma = \{x \mapsto p_i\}$ and $i \in \{1,\ldots,m\}$
(case eval)	$(f)case\ e\ of\ \{\overline{p_m \to e_m}\}$	$\Longrightarrow_{\sigma}\ \sigma((f)case\ e'\ of\ \{\overline{p_m \to e_m}\})$ if e is not in head normal form and $e \Longrightarrow_{\sigma} e'$
(fun)	$f(\overline{t_n})$	$\Longrightarrow_{id}\ \sigma(e)$ if $f(\overline{x_n}) = e \in \mathscr{R}$ and $\sigma = \{\overline{x_n \mapsto t_n}\}$

Fig. 3. Standard operational semantics (LNT calculus).

Observe that there is no rule to evaluate a rigid case expression with a variable argument. This situation produces a *suspension* of the evaluation.

The case eval rule can be applied when the argument of the case construct is not in head normal form (i.e., it is either a function call or another case construct). Then, it tries to evaluate this expression recursively.

Finally, the fun rule performs the unfolding of a function call. As in proof procedures for logic programming, we assume that we take a program rule with fresh variables in each such evaluation step.

Note that there is no rule to evaluate terms in head normal form; in this case, the computation stops *successfully*. An LNT *derivation* is denoted by $e_0 \Longrightarrow_{\sigma}^{*} e_n$, which is a shorthand for the sequence $e_0 \Longrightarrow_{\sigma_1} \ldots \Longrightarrow_{\sigma_n} e_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). An LNT derivation $e \Longrightarrow_{\sigma}^{*} e'$ is *successful* when e' is in head normal form. Then, we say that e *evaluates* to e' with *computed answer* σ.

Example 5

Consider the function "$\leqslant$" of Example 4. Given the initial call "`(Succ x)` $\leqslant$ `y`", the LNT calculus computes, among others, the following successful derivation:

```
(Succ x) ⩽ y
⟹id   fcase (Succ x) of                                          (fun)
         {Z       → True;
         (Succ n) → fcase y of {Z → False; (Succ m) → n ⩽ m}}
⟹id   fcase y of {Z → False; (Succ m) → x ⩽ m}                    (select)
⟹{y↦Z} False                                                      (guess)
```

Therefore, `(Succ x)` $\leqslant$ `y` evaluates to `False` with computed answer $\{\mathtt{y} \mapsto \mathtt{Z}\}$.

3 Forward slicing

In this section, we formalize our notion of *forward slice* in the context of functional logic programs. As mentioned before, in our setting any function call may play the role of *slicing criterion*. Essentially, given a program $\mathscr{R}$ and a (partially instantiated) call t – the slicing criterion – an associated forward slice is a *fragment* of $\mathscr{R}$ which contains all the statements which are necessary for executing the call t, i.e., which are *needed* to evaluate the slicing criterion. This relation between neededness – in the sense of Huet and Lévy (1992) – and slicing is not new; indeed, there exist several

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

approaches to slicing of functional programs which rely on the computation of neededness information (Biswas 1997; Field and Tip 1998). Clearly, t must compute the same value in $\mathscr{R}$ and in the computed slice. In particular, the original program is always a correct slice w.r.t. any slicing criterion. Our aim is thus to find smaller slices.[4] Furthermore, we do not distinguish between dynamic and static slicing, since it only depends on the degree of instantiation of the slicing criterion; in order words, we consider a sort of quasi static slicing (Venkatesh 1991).

As mentioned before, we do not consider the construction of amorphous slices; otherwise, partial evaluation could straightforwardly be seen as a slicing technique. Here, we only allow the deletion of some elements of the original program:

Term deletion: This is the simplest kind of deletion. It consists of the removal of subterms which are not needed to perform computations with the slicing criterion.

Branch deletion: By using the partially known data in the slicing criterion, some case branches become useless and can be deleted.

Function deletion: Finally, those functions which are not necessary to evaluate the slicing criterion can be completely deleted from the slice.

Analogously to Schoening and Ducassé (1996), our notion of *program slice* is formalized in terms of an abstraction relation. In the following, we consider that program signatures are implicitly augmented with the 0-ary constructor $\top$, a special symbol which is used to denote that some code fragment is missing.

Definition 1 (*term abstraction*)
A term t' is an abstraction of term t, in symbols $t' \geq t$, iff $t' = \top$ or $t' = t$.

Definition 2 (*expression abstraction*)
An expression e' is an abstraction of an expression e, in symbols $e' \geq e$, iff one of the following conditions holds:

- $e' = \top$ (i.e., a case structure is completely deleted);
- $e' = e$;
- $e' = (f)case\ x\ of\ \{\overline{p'_n \to e'_n}\}$, $e = (f)case\ x\ of\ \{\overline{p_n \to e_n}\}$, and $e'_i \geq e_i$ for all $i = 1, \ldots, n$.

Definition 3 (*program slice*)
A program $\mathscr{R}' = (D'_1, \ldots, D'_m)$ is a slice of a program $\mathscr{R} = (D_1, \ldots, D_m)$, in symbols $\mathscr{R}' \geq \mathscr{R}$, iff for all $i = 1, \ldots, m$, $D'_i = (f(\overline{x_n}) = e')$, $D_i = (f(\overline{x_n}) = e)$, and $e' \geq e$.

Roughly speaking, a program $\mathscr{R}'$ is a slice of program $\mathscr{R}$ if it can be obtained by replacing some subterms, case branches, and right-hand sides of function definitions by $\top$. Trivially, program slices are steadily executable (and fulfill the syntax of Figure 2) by just considering $\top$ as an arbitrary constant of the program's signature. 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).

[4] Weiser proved that computing the *minimal* slice is generally undecidable (Weiser 1984).

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

```
main op xs = fcase op of {Len → fst (lenmax xs);
                          Max → snd (lenmax xs)}
lenmax xs = (len xs, max xs)
len xs = fcase xs of {[] → Zero;
                      (x:xs) → Succ (len xs)}
max xs = fcase xs of {(y:ys) → fcase ys of
                                 {[] → y;
                                 (z:zs) → if (y ⩽ z) then max (z:zs)
                                                     else max (y:zs) }}
x ⩽ y = fcase x of {Zero   → True;
                    Succ n → fcase y of {Zero   → False;
                                         Succ m → n ⩽ m}}
fst x = fcase x of {(a,b) → a}
snd x = fcase x of {(a,b) → b}
```

Fig. 4. Example `lenmax`.

So far, we have only considered the *shape* of a slice. Now, we consider the *semantics* of the slicing process:

Definition 4 (*correct slice*)
Let $\mathscr{R}$ be a program and t a term. We say that $\mathscr{R}'$ is a correct slice of $\mathscr{R}$ w.r.t. t iff

- $\mathscr{R}'$ is a program slice of $\mathscr{R}$ (i.e., $\mathscr{R}' \succeq \mathscr{R}$), and
- $t \Longrightarrow^*_{\sigma_1} t_1$ in $\mathscr{R}$ iff $t \Longrightarrow^*_{\sigma_2} t_2$ in $\mathscr{R}'$, where t_1, t_2 are values (different from $\top$), $t_2 \succeq t_1$, and $\sigma_1 = \sigma_2$ (modulo variable renaming).

Observe that evaluations in the slice may produce values with some occurrences of $\top$ at inner positions, which is safe in our context since only the outermost symbol is *observable* in the LNT semantics. On the other hand, no abstraction is needed for substitutions, since the computed bindings can only map variables to patterns of the form $c(\overline{x_n})$ with no occurrences of $\top$ (see rule guess in Figure 3).

Example 6
Consider the program excerpt shown in Figure 4 for computing the length or the maximum of a list, depending on the value of the first parameter of `main`. Standard functions "`len`", "`max`", "`fst`", and "`snd`" return the length of a list, the maximum of a list, the first element of a tuple, and the second element of a tuple, respectively. Given the slicing criterion "`main Len xs`", the following slice can be obtained:

```
main op xs = fcase op of {Len → fst (lenmax xs);
                          Max → ⊤}
lenmax xs = (len xs, ⊤)
len xs = fcase xs of {[]     → Zero;
                      (x:xs) → Succ (len xs)}
max xs = ⊤
x ⩽ y = ⊤
fst x = fcase x of {(a,b) → a}
snd x = ⊤
```

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Here, we have performed three different kinds of code deletion:

Term deletion: The evaluation of the call to function "`max`" in the right-hand side of "`lenmax`" is not needed – since function "`fst`" only demands the evaluation of the first component of the tuple – and, thus, it has been replaced by $\top$.

Branch deletion: In the definition of function "`main`", the second branch of the case expression is not needed to execute the slicing criterion; therefore, it has also been replaced by $\top$.

Function deletion: Since functions "`max`", "$\leqslant$", and "`snd`" are no longer necessary to evaluate the slicing criterion, their definitions have been replaced by $\top$.

Note that this slice could not be constructed by using a simple graph of functional dependencies (e.g., functions "`snd`", "`lenmax`", and "$\leqslant$" depend on function "`main`" but they do not appear in the computed slice).

In order to simplify the representation of program slices, in the following we adopt the following conventions:

- case branches of the form $p \to \top$ are deleted and
- function definitions of the form $f(\overline{x_n}) = \top$ are removed from the slice.

Therefore, the slice of Example 6 is simply written as follows:

```
main op xs = fcase op of {Len → fst (lenmax xs)}
lenmax xs = (len xs, ⊤)
len xs = fcase xs of {[]     → Zero;
                      (x:xs) → Succ (len xs)}
fst x = fcase x of {(a,b) → a}
```

4 Monovariant/monogenetic partial evaluation

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 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 program rather than a specialized version.

In this section, we first recall the basic narrowing-driven partial evaluation (NPE) scheme (Albert and Vidal 2002) and, then, modify it in order to obtain a monovariant and monogenetic partial evaluator.

Essentially, NPE 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 (Alpuente *et al.* 1998), 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 (1993) for the partial evaluation of logic programs.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

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

Fig. 5. Narrowing-driven partial evaluation procedure.

The computation of a closed set of expressions can be regarded as the construction of a graph containing 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 5 (*closedness*)
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;
- $e = (f)case\ e'\ of\ \{\overline{p_m \to e_m}\}$ is a case expression and $e', e_1, \ldots, e_m$ are recursively E-closed;
- e is operation-rooted, there is an expression $e' \in E$, a matching substitution σ with $e = \sigma(e')$, and, for all $x \mapsto e'' \in \sigma$, e'' is recursively E-closed.

The basic partial evaluation procedure is shown in Figure 5. Let us explain the operators in this procedure:

- The operator *unfold* takes a program and a set of expressions $E_i = \{e_1, \ldots, e_n\}$, computes a *finite* set of (possibly incomplete) finite derivations $e_j \Longrightarrow^* e'_j$, $j = 1, \ldots, n$, and returns the set of derived expressions $E' = \{e'_1, \ldots, e'_n\}$. Here, partial computations are performed with the LNT calculus of Figure 3 slightly extended to avoid the backpropagation of bindings: the RLNT (for Residualizing LNT) calculus of Albert *et al.* (2003). The main difference between the LNT and the RLNT calculi is that the non-deterministic rule guess of the LNT calculus is replaced by a deterministic rule that leaves the case structure untouched and proceeds with the evaluation of the branches.
- Function *abstract* is then used to properly add the new expressions to the current set of (to be) partially evaluated expressions. For instance, a trivial abstraction operator could be defined as follows:

$$abstract(E_i, E') = E_i \cup \{e \in E' \mid \text{ there is no } e' \in E_i \text{ such that } e = e'\}$$

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Here, only the new expressions that are not equal (modulo variable renaming) to some expression in the current set E_i are added. This abstraction operator, however, does not guarantee the termination of the process since an infinite number of different expressions can be derived. In general, a termination test is also applied, e.g., Alpuente *et al.* (1998) consider a variant of the Kruskal tree condition called "homeomorphic embedding" (Leuschel 2002): if an expression embeds another expression in the current set, some form of generalization – usually the *most specific generalization* operator – is applied and the generalized term is added to the current set.

- 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 as follows:

$$build_residual_program(E_i, \mathscr{R}) = \{e = e' \mid e \in E_i \text{ and } e \Longrightarrow^* e' \text{ in } \mathscr{R}\}$$

In general, this operator also applies a renaming of expressions and some post-unfolding transformations which are not relevant for this work; we refer the interested reader to Albert and Vidal (2002).

In principle, the NPE scheme has been designed to achieve both polyvariant and polygenetic specializations. In this work, however, we are interested in the definition of a less powerful monovariant and monogenetic scheme. For this purpose, we should impose several restrictions to the procedure of Figure 5:

1. Firstly, the current set E_i should only contain 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 and, thus, we do not produce residual functions that mix several functions of the original program.
2. Secondly, the unfolding operator should perform only a one-step evaluation of each call rather than a computation of an arbitrary length. This condition is required to guarantee that no reachable function is hidden by the unfolding process. For instance, if we would allow a computation of the form `f x` $\Longrightarrow$ `g x` $\Longrightarrow$ `h x`, the unfolding operator would only return `h x`, while `g x` should also be part of the slice.
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, i.e., that only one residual definition is produced (at most) for each original function.

Unfortunately, such a monovariant/monogenetic partial evaluator would propagate information poorly. In order to overcome this drawback, in the next section we introduce a carefully designed operational mechanism which avoids the loss of information (i.e., program dependences) as much as possible.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

5 Computing program dependences

In this section, we introduce the kernel of a monovariant and monogenetic partial evaluator that can be used to compute program dependences. In principle, such a partial evaluator could proceed as follows:

- terms containing nested function symbols are *flattened*;
- terms in the current set of (to be) partially evaluated terms which are rooted by the same function symbol are generalized with some appropriate generalization operator (e.g., the *most specific generalization* operator).

However, flattening terms with nested function symbols would imply a serious loss of precision. For instance, a term of the form "`fst (lenmax xs)`" would be replaced by the terms "`fst y`" and "`lenmax xs`", where `y` is a fresh variable, thus missing the fact that `fst` is called with the result of "`lenmax xs`".

In order to avoid this loss of precision, we drop the first restriction above, i.e., we consider arbitrary operation-rooted terms during partial evaluation. However, we should still ensure that only a one-step of unfolding is applied to each term in order to guarantee that no reachable function is hidden by the unfolding process. In our flat language, function calls are evaluated *lazily*: a term 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, "`fst (lenmax xs)`" is unfolded to

```
fcase (lenmax xs) of {(a,b) → a}
```

Then, the evaluation of function "`fst`" cannot continue until the inner call to "`lenmax`" is reduced to a value. Unfortunately, this interleaved evaluation is problematic in our context since it would give rise to a polygenetic partial evaluation (i.e., a residual function comprising the evaluation of both `fst` and `lenmax`). 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.

For this purpose, we extend the partial evaluation mechanism in order to work on *states* rather than on expressions.

Definition 6 (*state*)
A state is a pair of the form $\langle e, S \rangle$, where e is an expression (to be evaluated) and S is a *stack* (a list) which represents the current "evaluation context".[5] The empty stack is denoted by [ ].

For example, the previous expression "`fst (lenmax xs)`" could now be flattened as follows (see Example 7): ⟨`lenmax xs`, [(`fst x,x`)]⟩, which means that `lenmax xs` is ready to perform a complete one-step unfolding; when this evaluation is performed, the initial term can be reconstructed thanks to the information in the stack, (`fst x,x`), which means that the initial term has the form `fst x`, where `x` is

[5] Similar operational semantics with a stack can be found in Albert *et al.* (2005) and Sestoft (1997).

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

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

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

Fig. 6. Extended operational semantics.

the result of evaluating the first component of the state (i.e., the result of evaluating `lenmax xs`). Thanks to the use of states, we do not miss the fact that `fst` is called with the result of "`lenmax xs`".

Figure 6 shows an extended operational semantics which is appropriate to deal with states. Let us briefly explain the rules of this operational semantics.

Rules select and guess proceed in a similar way as their counterpart in the standard semantics of Figure 3.

Rule flatten is used to avoid the unfolding of those (operation-rooted) terms whose unfolding would demand the evaluation of some inner call. This is necessary to ensure that partial evaluation is monogenetic. In this case, we delay the function unfolding and continue with the evaluation of 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).

Example 7

Consider again the program of Example 6. In order to flatten the following expression: $\langle$`fst (lenmax xs), []`$\rangle$, we proceed as follows. First, we perform a function unfolding so that we get:

$\langle$`fcase (lenmax xs) of {(a,b) → a}, []`$\rangle$

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Now, we try to evaluate this state by means of rules select and guess. Since no reduction is possible and the case structure has a function call in the argument position, function *flat* returns the state

$$\langle \texttt{lenmax xs}, [(\texttt{fst x}, \texttt{x})] \rangle$$

where x is a fresh variable. Observe that this state cannot be further flattened since a function unfolding returns the state

$$\langle (\texttt{len xs}, \top), [\,] \rangle$$

which cannot be reduced by rules select and guess and which contains no function call in the argument position of a case expression. Therefore, in this case, function *flat* returns $\bot$ and no step with rule flatten can be done.

Rule fun performs a simple function unfolding when rule flatten does not apply, i.e., when 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 of Figure 3. There are, though, the following main differences:

- Now, the one-step relation $\Longrightarrow$ is not labeled with the computed bindings since we are not interested in computing answers but only in obtaining the functions which are reachable from the initial call.
- In the standard semantics, rigid case expressions with a free variable in the argument position *suspend*. 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 explore all the alternatives of rigid case expressions.
- The order of evaluation is slightly 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. Roughly speaking, the extended semantics is in between the standard operational semantics and its residualizing version used to perform partial computations in the NPE framework (Albert *et al.* 2003).

Example 8

Consider again the program of Example 6. Given the initial term `fst (lenmax xs)`, we have (among others) the following (incomplete) computation with the standard semantics of Figure 3:

$$\begin{array}{lll}
\texttt{fst (lenmax xs)} & \Longrightarrow_{\texttt{id}} \ \texttt{fcase (lenmax xs) of \{(a,b)} \to \texttt{a\}} & \text{(fun)} \\
& \Longrightarrow_{\texttt{id}} \ \texttt{fcase (len xs, max xs) of \{(a,b)} \to \texttt{a\}} & \text{(fun)} \\
& \Longrightarrow_{\texttt{id}} \ \texttt{len xs} & \text{(select)}
\end{array}$$

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

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

Fig. 7. Computation of reachable program points.

On the other hand, the extended operational semantics of Figure 6 performs the following equivalent derivation:

$$\begin{array}{lcll}
\langle \texttt{fst (lenmax xs)}, [\,]\rangle & \Longrightarrow & \langle \texttt{lenmax xs}, [\texttt{(fst x,x)}]\rangle & \text{(flatten)}\\
& \Longrightarrow & \langle \texttt{(len xs, max xs)}, [\texttt{(fst x,x)}]\rangle & \text{(fun)}\\
& \Longrightarrow & \langle \texttt{fst (len xs, max xs)}, [\,]\rangle & \text{(replace)}\\
& \Longrightarrow & \langle \texttt{fcase (len xs, max xs) of } \{\texttt{(a,b)} \to \texttt{a}\}, [\,]\rangle & \text{(fun)}\\
& \Longrightarrow & \langle \texttt{len xs}, [\,]\rangle & \text{(select)}
\end{array}$$

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 unfolding of some function call. From these sequences, a monogenetic/monovariant partial evaluation scheme can easily be defined and, thus, the algorithm for computing dependences in our program slicing technique.

The algorithm of Figure 5 is now slightly modified in order to work with states. The new algorithm (depicted in Figure 7) does not compute a residual program but only the set of states which are reachable from the initial call. In other words, it returns the counterpart of the final set of *closed* terms computed by the algorithm of Figure 5. The new algorithm starts by flattening the initial term to ensure that a complete one-step unfolding can be performed. We now tackle the definition of appropriate unfolding and abstraction operators. First, the *one-step* unfolding operator is defined as follows:

Definition 7 (*unfold*)
Let $\mathscr{S}$ be a set of states. The unfolding operator *unfold* is defined by

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

where

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

This unfolding operator always performs a complete one-step unfolding of each input expression. The associated stack S remains unchanged since only rules flatten

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

and replace can modify the current stack. Function *unf* returns a *set* of derived states because of the non-determinism of the underlying operational semantics.

Example 9
Consider again the program of Example 6. We illustrate function *unf* by means of some simple examples:

$$unf(\langle \texttt{lenmax xs}, [\texttt{(fst x,x)}]\rangle) = \langle \texttt{(len xs, max xs)}, [\texttt{(fst x,x)}]\rangle$$

$$unf(\langle \texttt{fst (len xs, max xs)}, [\,]\rangle) = \langle \texttt{len xs}, [\,]\rangle$$

according to the partial computation in Example 8.

Before defining our abstraction operator, we need the following auxiliary notion:

Definition 8 (*flattened state*)
Let s be a state returned by the 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 result:

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.

To add new states to the current set of states, we introduce the following abstraction operator:

Definition 9 (*abstract*)
Let $\mathscr{S}$ and $\mathscr{S}' = \{s_1, \ldots, s_n\}$ be sets of states. Our abstraction operator proceeds as follows: $abstract(\mathscr{S}, \mathscr{S}') = abs(abs(\ldots abs(\mathscr{S}, s'_1)\ldots, s'_{n-1}), s'_n)$, where:

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

Basically, function *abstract* starts by flattening the input states by applying (zero or one step of) rule replace, followed by (zero or more steps of) rule flatten.

Definition 10 (*abs*)
Function *abs* is defined inductively on the structure of flattened states (according to Lemma 1):

$$abs(\mathscr{S}, \langle x, [\,]\rangle) = \mathscr{S}$$

$$abs(\mathscr{S}, \langle c(\overline{t_n}), [\,]\rangle) = abstract(\mathscr{S}, \mathscr{S}')$$
if $\overline{t'_m}$ are the maximal operation-rooted subterms of $c(\overline{t_n})$ and $\mathscr{S}' = \{\overline{\langle t'_m, [\,]\rangle}\}$

$$abs(\mathscr{S}, \langle f(\overline{t_n}), S\rangle) = \begin{cases} \mathscr{S} \cup \{\langle f(\overline{t_n}), S\rangle\} & \text{if there is no state } \langle f(\overline{t'_n}), S'\rangle \text{ in } \mathscr{S} \\ \mathscr{S} & \text{else if } \langle f(\overline{t_n}), S\rangle \text{ is } \mathscr{S}\text{-closed} \\ abstract(\mathscr{S}^*, \mathscr{S}'') & \text{otherwise, where } \langle f(\overline{t'_n}), S'\rangle \in \mathscr{S}, \\ & msg(\langle f(\overline{t'_n}), S'\rangle, \langle f(\overline{t_n}), S\rangle) = (\langle f(\overline{t''_n}), S'\rangle, \mathscr{S}''), \\ & \text{and } \mathscr{S}^* = (\mathscr{S} \setminus \{\langle f(\overline{t'_n}), S'\rangle\}) \cup \{\langle f(\overline{t''_n}), S'\rangle\} \end{cases}$$

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

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. 5), 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 σ_1 and σ_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.

Example 10

Consider the set of states $\mathcal{S} = \{\langle \mathtt{len\ xs}, [\,] \rangle, \langle \mathtt{fst\ (a,b)}, [\,] \rangle\}$. We illustrate function (*abs*) by means of some simple examples:

$$abs(\mathcal{S}, \langle \mathtt{max\ xs}, [\,] \rangle) = \mathcal{S} \cup \{\langle \mathtt{max\ xs}, [\,] \rangle\}$$

since there is no state rooted by function `max` in $\mathcal{S}$,

$$abs(\mathcal{S}, \langle \mathtt{len\ (y:ys)}, [\,] \rangle) = \mathcal{S}$$

since $\langle \mathtt{len\ (y:ys)}, [\,] \rangle$ is $\mathcal{S}$-closed (i.e., `len (y:ys)` is an instance of `len xs`), and

$$abs(\mathcal{S}, \langle \mathtt{fst\ z}, [\,] \rangle) = \{\langle \mathtt{len\ xs}, [\,] \rangle, \langle \mathtt{fst\ w}, [\,] \rangle\}$$

since there is a state $\langle \mathtt{fst\ (a,b)}, [\,] \rangle$ rooted by function `fst`, the state $\langle \mathtt{fst\ z}, [\,] \rangle$ is not $\mathcal{S}$-closed (since `fst z` is not an instance of `fst (a,b)`, the most specific

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

generalization of the states $\langle$`fst (a,b)`$\rangle$ and $\langle$`fst z,[]`$\rangle$ returns $\langle$`fst w,[]`$\rangle$, and

$$abs(\{\langle \texttt{len xs},[]\rangle\},\langle \texttt{fst w},[]\rangle) = \{\langle \texttt{len xs},[]\rangle,\langle \texttt{fst w},[]\rangle\}$$

since there is no state in $\{\langle$`len xs,[]`$\rangle\}$ rooted by function `fst`.

Our operator *abstract* can be seen as an instance of the parametric abstraction operator introduced by Alpuente *et al.* (1998) particularized to consider states (rather than terms) and monovariant partial evaluation (thus, only one operation-rooted term is allowed for each defined function symbol). Our 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. $abstract(\mathcal{S},\mathcal{S}')$.

This lemma is a crucial result to ensure the correctness of our approach. In fact, it will allow us to prove that the generated program is a correct slice according to Definition 4.

Example 11
Consider again the program of Example 6. Given the slicing criterion "`main Len xs`", the initial set of states is $\mathcal{S}_0 = \{\langle$`main Len xs,[]`$\rangle\}$. Now, we show the sequence of iterations performed by the algorithm of Figure 7:

$$\begin{array}{l}
\mathcal{S}'_0 = \{\langle \texttt{fst (lenmax xs),[]}\rangle\}\\
\mathcal{S}_1 = \mathcal{S}_0 \cup \{\langle \texttt{lenmax xs, [(fst x,x)]}\rangle\}\\
\mathcal{S}'_1 = \mathcal{S}'_0 \cup \{\langle \texttt{(len xs,max xs), [(fst x,x)]}\rangle\}\\
\mathcal{S}_2 = \mathcal{S}_1 \cup \{\langle \texttt{fst (len xs, max xs),[]}\rangle\}\\
\mathcal{S}'_2 = \mathcal{S}'_1 \cup \{\langle \texttt{len xs,[]}\rangle\}\\
\mathcal{S}_3 = \mathcal{S}_2 \cup \{\langle \texttt{len xs,[]}\rangle\}\\
\mathcal{S}'_3 = \mathcal{S}'_2 \cup \{\langle \texttt{Zero,[]}\rangle, \langle \texttt{Succ (len xs),[]}\rangle\}\\
\mathcal{S}_4 = \mathcal{S}_3
\end{array}$$

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

$$\begin{array}{l}
\mathcal{S} = \{\langle \texttt{main Len xs,[]}\rangle, \langle \texttt{lenmax xs, [(fst x,x)]}\rangle,\\
\qquad \langle \texttt{fst (len xs, max xs),[]}\rangle, \langle \texttt{len xs,[]}\rangle \quad \}
\end{array}$$

The total correctness of the algorithm in Figure 7 is stated in the following theorem:

Theorem 1
Given a flat program $\mathcal{R}$ and an initial term t, the algorithm in Figure 7 terminates computing a set of states $\mathcal{S}$ such that $\langle t,[]\rangle$ is $\mathcal{S}$-closed.

6 Extraction of the slice

In this section, we introduce the final step of our slicing process, i.e., the extraction of the program slice. Let us recall that it must be a *fragment* of the original

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Rule	*Expr*		*Expr*
var	$[\![x]\!]\rho$	$\longrightarrow$	x
cons	$[\![c(\overline{t_n})]\!]\rho$	$\longrightarrow$	$c([\![t_1]\!]\rho,\ldots,[\![t_n]\!]\rho)$
select	$[\![(f)case\ x\ of\ \{\overline{p_k \to e_k}\}]\!]\rho$	$\longrightarrow$	$(f)case\ x\ of\ \{p_i \to [\![e_i]\!]\rho'\}$
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: $\rho(x) \in \mathscr{X}$, $\rho_i = \{x \mapsto p_i\} \circ \rho$, and $i \in \{1,\ldots,k\}$
fun: there is some term in *residual_calls*($\mathscr{S}$) rooted by f
remove: otherwise

Fig. 8. Simplified unfolding rules.

program – thus no instantiation of variables is allowed – and produce the same outputs for the slicing criterion as the original program. Here, we follow the *simplified* form for program slices, i.e., case branches of the form $p \to \top$ are deleted, and function definitions of the form $f(\overline{x_n}) = \top$ do not appear in the slice.

First, we need the following auxiliary function that returns the terms which are relevant in order to extract a program slice from the set of states computed by the algorithm of Figure 7:

Definition 11 (*residual calls*)
Let $\mathscr{S}$ be a set of states returned by the algorithm of Figure 7 and let $T_{\mathscr{S}} = \{t \mid \langle t, S\rangle \in \mathscr{S}\}$. Then, the set of residual calls of $\mathscr{S}$ is defined as follows:

$$residual_calls(\mathscr{S}) = T_{\mathscr{S}} \cup \{t' \mid \langle t, S\rangle \in \mathscr{S}, t' \in calls(S), \text{and } t' \text{ is not } T_{\mathscr{S}}\text{-closed}\}$$

Observe that, in the above definition, *residual_calls* should also return the function calls in the computed stacks when they are not closed w.r.t. the set of first components of the states in $\mathscr{S}$. This is mandatory in order to ensure a full equivalence w.r.t. the standard semantics. Program slices can now be built as follows:

Definition 12 (*construction of program slices*)
Let $\mathscr{S}$ be a set of states returned by the algorithm of Figure 7. Then, a program slice is obtained from *build_slice*(*residual_calls*($\mathscr{S}$)), where function *build_slice* is defined as follows:

$$\begin{array}{lcl} build_slice(\{\,\}) & = & \{\,\} \\ build_slice(\{f(\overline{t_n})\} \cup T') & = & \{f(\overline{x_n}) = e'\} \cup build_slice(T') \\ & & \text{where } f(\overline{x_n}) = e \in \mathscr{R},\ \rho = \{\overline{x_n \mapsto t_n}\}, \\ & & \text{and } [\![e]\!]\rho \longrightarrow^* e' \not\longrightarrow \end{array}$$

The new calculus which is used to construct the rules of the slice is depicted in Figure 8. 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 ρ is used to store the bindings for the program variables. Let us briefly explain the rules of the new calculus.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Rule var simply returns a variable unchanged. Rule cons applies to constructor-rooted terms; it leaves the outermost constructor symbol and (recursively) inspects the arguments.

Rules select and guess proceed similarly to their counterpart in Figure 6 but leave the case structure untouched; the substitution ρ 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 (cf. Figure 2). Note that rule guess is now deterministic (and, thus, the entire calculus).

Finally, rules fun and remove are used to reduce function calls: when there is some term in $\mathit{residual_calls}(\mathscr{S})$ with the same outermost function symbol, we proceed as in rule cons; otherwise, we return $\top$ (which means that the evaluation of this function call is not needed).

Example 12
Consider the set of states computed in Example 11. From this set, function *residual_calls* returns the set of terms:

$$\{\texttt{main Len xs}, \texttt{lenmax xs}, \texttt{fst (len xs, max xs)}, \texttt{len xs}\}$$

Now, we construct a residual rule for each term of the set. For instance, for the term "`main Len xs`", the associated residual rule is:

$$\texttt{main op xs} = \texttt{fcase op of } \{\texttt{Len} \to \texttt{fst (lenmax xs)}\}$$

since the following derivation can be performed (with $\rho = \{\texttt{op} \mapsto \texttt{Len}\}$):

$$\begin{aligned}
&[\![\texttt{fcase op of } \{\ \texttt{Len} \to \texttt{fst (lenmax xs)};\ \texttt{Max} \to \texttt{snd (lenmax xs)}\ \}]\!]\rho \\
&\quad \longrightarrow_{\mathsf{select}} \texttt{fcase op of } \{\texttt{Len} \to [\![\texttt{fst (lenmax xs)}]\!]\rho\} \\
&\quad \longrightarrow_{\mathsf{fun}} \texttt{fcase op of } \{\texttt{Len} \to \texttt{fst } ([\![\texttt{lenmax xs}]\!]\rho)\} \\
&\quad \longrightarrow_{\mathsf{fun}} \texttt{fcase op of } \{\texttt{Len} \to \texttt{fst (lenmax } [\![\texttt{xs}]\!]\rho)\} \\
&\quad \longrightarrow_{\mathsf{var}} \texttt{fcase op of } \{\texttt{Len} \to \texttt{fst (lenmax xs)}\}
\end{aligned}$$

By constructing a residual rule associated to each of the remaining terms, the computed slice coincides with the (simplified version of the) program slice which is shown in Example 6.

Now, we show that the result of Definition 12 is a program slice of the original program according to Definition 3.

Theorem 2
Let $\mathscr{R}$ be a flat program and t a term. Let $\mathscr{S}$ be a set of states computed by the algorithm of Figure 7 from $\mathscr{R}$ and t. Then, $\mathscr{R}' = \mathit{build_slice}(\mathit{residual_calls}(\mathscr{S}))$ is a program slice of $\mathscr{R}$, i.e., $\mathscr{R}' \succeq \mathscr{R}$.

Finally, the correctness of the computed slices (according to Definition 4) is inherited by the correctness of the underlying partial evaluation process.

Theorem 3
Let $\mathscr{R}$ be a flat program and t a term. Let $\mathscr{S}$ be a set of states computed by the algorithm of Figure 7 from $\mathscr{R}$ and t. If computations for t in $\mathscr{R}$ do not suspend,

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Table 1. *Partial evaluator vs program slicer – code structure*

		`main`	`global`	`local`	`post`	`util`	**Total**
Partial evaluator	*(lines)*	306	403	888	433	316	**2346**
	(functions)	22	43	83	44	38	**230**
Program slicer	*(lines)*	232	486	249	195	419	**1581**
	(functions)	20	50	29	26	55	**180**

then t computes the same values and answers in $\mathscr{R}$ and in $build_slice(residual_calls(\mathscr{S}))$.

7 Implementation

To check the practicality of the ideas presented so far, a prototype implementation of the program slicer for Curry programs has been developed in Curry itself. The resulting tool covers not only the flat programs of Section 2 but also source Curry programs (which are automatically translated to the flat syntax). Moreover, it also accepts higher-order functions, overlapping left-hand sides, several predefined (built-in) functions, etc. The implemented tool is publicly available from `http://www.dsic.upv.es/users/elp/german/slicing/`.

It is worthwhile to note that the development of the program slicer required a small implementation effort since it was developed by extending an existing partial evaluator for Curry programs (Albert *et al.* 2002). Table 1 shows the structure of both the partial evaluator and the program slicer, including the lines of code and the number of functions for each basic component:

`main`: basic definitions and data type declarations, reading of source program, writing of transformed program, etc;
`global`: global control, including termination tests and generalization operations;
`local`: local control, i.e., a non-standard meta-interpreter;
`post`: post-processing transformation, i.e., renaming and post-unfolding compression in the partial evaluator and extraction of the slice in the program slicer;
`util`: general utilities and pretty printing.

Basically, components `main`, `local`, and `util` were almost straightforwardly adapted from the partial evaluator to the program slicer. For instance, component `local` of the program slicer – which corresponds to the semantics shown in Figure 6 – is a simplified version of the same component in the partial evaluator, since only a one-step unfolding is required here. More significant changes were made in component `global`. In contrast to the partial evaluator, the program slicer introduces the use of *states* and, thus, it required the implementation of rules replace and flatten, as well as the associated abstraction operator. Finally, component `post` of the partial evaluator was entirely replaced, since the program slicer does not perform neither renaming nor post-unfolding compression but should only extract the residual rules according to the calculus of Figure 8.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Our slicing tool is able to compute the slice of Example 6, thus it is strictly more powerful than naive approaches based on graphs of functional dependences. In general, forward slicing has been proved particularly useful in the areas of program understanding, dead code removal, and code reuse. Now, we illustrate the application of the program slicer with some selected examples. First, we consider the program of Example 6 (in Curry syntax):

```
main Len xs = fst (lenmax xs)
main Max xs = snd (lenmax xs)

lenmax xs = (len xs, max xs)

len []     = Z
len (x:xs) = Succ (len xs)

max [x]      = x
max (x:y:ys) = if (x ⩽ y) then max (y:ys)
                          else max (x:ys)
Z ⩽ m               = True
(Succ n) ⩽ Z        = False
(Succ n) ⩽ (Succ m) = n ⩽ m

fst (a,b) = a
snd (a,b) = b
```

Given the slicing criterion "`main Len xs`", our tool returns the following slice:

```
main Len xs = fst (lenmax xs)

lenmax xs = (len xs, ⊤)

len []     = Z
len (x:xs) = Succ (len xs)

fst (a,b) = a
```

Here, the second rule of function `main` as well as the definitions of functions `max`, $\leqslant$, and `snd` have been sliced away, since they are not needed when the first parameter of `main` is the constant `Len`. Note that the removal of *case branches* in the flat language is now viewed in Curry as the removal of *rules* in a function definition, e.g.,

```
main op xs = fcase op of {Len → fst (lenmax xs);
                          Max → ⊤}
```

is simply written as follows:

```
main Len xs = fst (lenmax xs)
```

Let us now consider a similar situation but in a higher-order context:

```
trans p xs = map (f p) xs

map f []     = []
map f (x:xs) = f x : map f xs
```

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

```
f A = inc
f B = dec
f C = square
...

inc x = Succ x
dec (Succ x) = x
square x = x * x
...
```

Function `trans` applies a parametric function, `f`, to all the elements of a given list. Now, the computed slice w.r.t. the slicing criterion "`trans A xs`" is as follows:

```
trans p xs = map (f p) xs

map f []     = []
map f (x:xs) = f x : map f xs

f A = inc

inc x = Succ x
```

Again, all functions but `inc` and the first rule for `f` have been deleted, which shows that our approach works well in the presence of higher-order functions. Finally, let us show an example which illustrates the removal of dead code due to lazy evaluation. Consider the following program:

```
lenInc n xs = len (incL n xs)

len []     = Z
len (x:xs) = Succ (len xs)

incL n []     = []
incL n (x:xs) = inc n : incL n xs

inc x = Succ x
```

Here, function `lenInc` takes a number and a list, and returns the length of the list which results from adding the given number to each element of the original list. Clearly, in a lazy context, function `inc` will never be executed. Therefore, the computed slice w.r.t. "`lenInc n xs`" (i.e., no input data is known) is as follows:

```
lenInc n xs = len (incL n xs)

len []     = Z
len (x:xs) = Succ (len xs)

incL n []     = []
incL n (x:xs) = ⊤ : incL n xs
```

The occurrence of ⊤ in the definition of `incL` shows that the values of the elements in the list are not needed to compute the length of the given list.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Table 2. *Partial evaluator vs program slicer – selected benchmarks*

Benchmark	PE time ms	Slicing time ms	Runtime			Size		
			Orig ms	Spec %	Sliced %	Orig bytes	Spec %	Sliced %
ackermann	20490	1370	1330	98.50%	99.25%	2039	228.3%	49.93%
allones	5090	1990	1140	111.40%	100.88%	3502	165.25%	63.42%
fibonacci	150	270	380	81.58%	102.63%	2438	64.93	50.82%
filtermap	280	450	1460	84.93%	100.00%	2147	28.50%	69.26%
fliptree	2800	1230	1430	92.31%	93.71%	2619	202.33%	52.20%
foldr.map	80	320	630	60.32%	100.00%	1784	21.41%	64.63%
foldr.sq	70	310	670	65.67%	101.49%	1763	21.10%	64.61%
foldr.sum	6730	1400	1570	80.25%	98.09%	4678	35.85%	13.92%
funinter	504930	2220	—	—	—	5288	657.19%	61.86%
gauss	11680	950	700	82.86%	98.57%	2115	61.56%	42.36%
iterate	1950	750	890	25.84%	103.37%	1968	117.99%	69.61%
kmpAAB	710	990	380	31.58%	105.26%	3348	42.29%	75.30%
kmpAAAAAAB	9870	3250	790	35.44%	100.00%	3968	104.86%	69.78%
power	12710	890	620	95.16%	103.23%	2830	203.29%	46.93%
quicksort	450	670	260	165.38%	103.85%	2711	84.06%	81.48%
reverse	4590	970	680	98.53%	101.47%	1873	251.09%	53.87%
Average	**36411**	**1127**	**862**	**80.65%**	**100.79%**	**2817**	**143.13%**	**58.12%**

Let us mention that, in contrast to the original partial evaluator, the implemented program slicer can deal with larger programs efficiently. This is mainly due to the monovariant/monogenetic nature of the underlying partial evaluator, which simplifies the computation of a closed set of terms. Table 2 shows a summary of the experiments conducted on an extensive set of benchmarks. We used the Curry→Prolog compiler of PAKCS 1.6.0 (Hanus *et al.* 2004) running on a 2.4 GHz Linux-PC (Intel Pentium IV with 512 KB cache). Runtime input goals were chosen to give a reasonably long overall time. Code size was obtained by measuring the intermediate FlatCurry files (suffix `.fcy`) generated by PAKCS. The considered benchmarks are available from `http://www.dsic.upv.es/users/elp/german/slicing/`.

The results in Table 2 show that the program slicer is in almost all cases much faster than the partial evaluation tool. As expected, the runtime of the sliced programs do not significantly differ from the runtimes of the original ones, since only some program rules (or expressions) have been deleted; this shows that little overhead has to be paid for adding extra functions to a program. Anyway, the main purpose of slicing is not speedup, but reducing code size. In this case, slicing has managed an overall code size reduction of 57.60% whereas the partial evaluator has increased the code size by 162.26%. Indeed, the slicing never increases the code size, while the partial evaluator has increased the code size by 657.18% in the worst case. On the other hand, there are cases where specialization achieves much smaller code size than slicing, e.g., for `filtermap` where the specializer has managed to transform the composition of several higher-order functions into a single first-order function.

8 Related work

Although program slicing was originally introduced in the imperative programming setting, it has been applied to almost all programming paradigms, e.g.,

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

object-oriented programs (Tip *et al.* 1996; Larsen and Harrold 1996; Steindl 1998), logic programs (Schoenig and Ducasse 1996; Zhao *et al.* 2001), functional programs (Biswas 1997; Field and Tip 1998), or algebraic specifications (Woodward and Allen 1998). Although we are not aware of any previous work addressing forward slicing of multi-paradigm functional logic programs, in the following we review the closest approaches to our work.

Within imperative programming, the closest approach is that of Blazy and Facon (1998), who use partial evaluation for program understanding in Fortran. Since they do not want to change the original structure of the code, no unfolding is performed (similarly to our one-step unfoldings). Also, they neither introduce new variables nor rename the existing ones. In this work, we have followed a very similar approach in order to define a forward slicing algorithm for functional logic programs. In both approaches, a simplified partial evaluator that does not change the structure of the original program has been introduced.

Within the logic programming paradigm, Gyimóthy and Paakki (1995) introduce the first approach to slicing. They define a specific slicing algorithm which computes a slice of the proof tree in order to reduce the number of questions asked by an algorithmic debugger (Shapiro 1983). The slice is computed from a static dependency graph containing only oriented data dependencies. In contrast to our work, their algorithm cannot be not used to compute *executable* programs. Schoening and Ducassé (1996) define the first (backward) slicing algorithm for Prolog programs which produce executable slices. They introduce an abstraction relation in order to formalize the notion of program slice. Our notion of slice in Section 3 is somehow inspired by this work. Leuschel and Sørensen (1996) introduce the concept of *correct erasure* to detect and remove redundant arguments from logic programs. They present a constructive algorithm for computing correct erasures which can be used to perform a simple form of slicing. Actually, Leuschel and Vidal (2005) have very recently introduced a new approach to forward slicing of logic programs which is based on a combination of the ideas presented in this work and the redundant argument filtering of Leuschel and Sørensen (1996).

As for functional programs, Field and Tip (1998) present a very detailed study of the concept of slicing associated with left-linear term rewriting systems (a notion of "program" very close to the one considered in our work). Their definition of slice is also based on a notion of *neededness* but, in contrast to our work, they consider backward slicing (and compute slices that are not executable on the standard interpreter). Another closely related approach has been introduced by Reps and Turnidge (1996). They define a backward slicing technique for functional programs which can be used to perform a sort of program specialization that cannot be achieved by standard partial evaluation. Their work can be seen as complementary to ours, since we are interested in the use of partial evaluation to perform program slicing. On the other hand, Hallgren (2003) reports some experiments with a Haskell slicer. It is mainly based on the construction of a graph of functional dependences and, thus, it is less powerful than our partial evaluation-based slicer.

Very recently, Ochoa *et al.* (2004) have introduced a novel approach to *dynamic* backward slicing of functional logic programs which is based on an extension of the tracing technique of Braßel *et al.* (2004). In particular, their approach relies on

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

constructing a *redex trail* of a given computation in order to compute all program dependences. Basically, a redex trail is a directed graph which records copies of all values and redexes of a computation, with a backward link from each reduct to the parent redex that created it. Then, a backward slice can easily be obtained by mapping the relevant nodes of the redex trail to concrete locations of the source program. This approach has also been applied to Haskell programs by Chitil (2004). These approaches are not based on partial evaluation but on well-known techniques for debugging functional programs. Therefore, the implementation of a *dynamic* slicer is relatively easy if one already has a debugger based on redex trails. However, they are not useful to develop a *static* slicing tool. In contrast, our approach can be used to perform both static and (forward) dynamic slicing.

9 Conclusions and future work

This work introduced the first approach to forward slicing of multi-paradigm (functional logic) programs. Although some extensions were needed, our developments basically rely on adapting and extending an online partial evaluation scheme for such programs. Thus, the implementation of an associated slicing tool was easily achieved by extending an existing partial evaluation tool. 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 (first-order) 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 (Sørensen *et al*. 1996). On the other hand, similar ideas have already been applied to define a forward slicing technique for logic programs (Leuschel and Vidal 2005).

An interesting topic for future work is the extension of our approach to perform *backward* slicing. Here, the computed slice should contain those program statements which are needed to compute some selected *fragment* of the output. While forward slicing is useful for program understanding, reuse, maintenance, etc., backward slicing can be applied to, e.g., program debugging, specialization and merging.

Acknowledgements

We gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2002 for many useful comments and suggestions. We also thank Michael Leuschel for his helpful remarks concerning the relation between partial deduction and slicing in the context of logic programming.

References

Albert, E., Hanus, M., Huch, F., Olvier, J. and Vidal, G. 2005. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation 40,* 1, 795–829.

Albert, E., Hanus, M. and Vidal, G. 2002. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming 2002,* 1.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Albert, E., Hanus, M. and Vidal, G. 2003. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs. *Information Processing Letters 85,* 1, 19–25.

Albert, E. and Vidal, G. 2002. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing 20,* 1, 3–26.

Alpuente, M., Falaschi, M. and Vidal, G. 1998. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS 20,* 4, 768–844.

Antoy, S. 1992. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*. Springer LNCS 632, 143–157.

Antoy, S., Echahed, R. and Hanus, M. 2000. A Needed Narrowing Strategy. *Journal of the ACM 47,* 4, 776–822.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That*. Cambridge University Press.

Biswas, S. 1997. Dynamic Slicing in Higher-Order Programming Languages. Ph.D. thesis, University of Pennsylvania.

Blazy, S. and Facon, P. 1998. Partial Evaluation for Program Comprehension. *ACM Computing Surveys 30,* 3es.

Brassel, B., Hanus, M., Huch, F. and Vidal, G. 2004. A Semantics for Tracing Declarative Multi-Paradigm Programs. In *Proc. of the 6th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'04)*. ACM Press, 179–190.

Chitil, O. 2004. Source-Based Trace Exploration. In *In Draft Proc. of the 16th Int'l Workshop on Implementation of Functional Languages (IFL 2004)*. Technical Report 0408, University of Kiel, 239–244.

Ferrante, J., Ottenstein, K. and Warren, J. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems 9,* 3, 319–349.

Field, J. and Tip, F. 1998. Dynamic Dependence in Term Rewriting Systems and its Application to Program Slicing. *Information and Software Technology 40,* 11–12, 609–634.

Gallagher, J. 1993. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*. ACM, New York, 88–98.

Giovannetti, E., Levi, G., Moiso, C. and Palamidessi, C. 1991. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences 42*, 363–377.

Glück, R. and Sørensen, M. 1996. A Roadmap to Metacomputation by Supercompilation. In *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, O. Danvy, R. Glück, and P. Thiemann, Eds. Springer LNCS 1110, 137–160.

Gyimóthy, T. and Paakki, J. 1995. Static Slicing of Logic Programs. In *Proc. of the 2nd Int'l Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*. IRISA-CNRS, 87–103.

Hallgren, T. 2003. Haskell Tools from the Programatica Project. In *Proc. of the 2003 ACM SIGPLAN Haskell Workshop*. ACM Press, 103–106.

Hanus, M. 1994. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming 19&20*, 583–628.

Hanus, M. 1997. A Unified Computation Model for Functional and Logic Programming. In *Proc. of ACM Symp. on Principles of Programming Languages (POPL'97)*. ACM, New York, 80–93.

Hanus, M. 2003. Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/ mh/curry/`.

Hanus, M., Antoy, S., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R. and Steiner, F. 2004. PAKCS 1.6.0: The Portland Aachen Kiel Curry System User Manual. Tech. rep., University of Kiel, Germany.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Hanus, M. and Prehofer, C. 1999. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming 9,* 1, 33–75.

Harman, M. and Danicic, S. 1997. Amorphous Program Slicing. In *Proc. of the 5th Int'l Workshop on Program Comprehension*. IEEE Computer Society Press.

Harman, M., Danicic, S. and Sivagurunathan, Y. 1995. Program Comprehension Assisted by Slicing and Transformation. In *Proc. of the 1st UK Program Comprehension Workshop*.

Harman, M. and Hierons, R. 2001. An Overview of Program Slicing. *Software Focus 2,* 3, 85–92.

Hortalá-González, T. and Ullán, E. 2001. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of the 5th Int'l Symp. on Functional and Logic Programming (FLOPS 2001)*. Springer LNCS 2024, 216–232.

Huet, G. and Lévy, J. 1992. Computations in orthogonal rewriting systems, Part I + II. In *Computational Logic – Essays in Honor of Alan Robinson*, J. Lassez and G. Plotkin, Eds. 395–443.

Jones, N., Gomard, C. and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ.

Klop, J. 1992. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Vol. I. Oxford University Press, 1–112.

Kuck, D., Kuhn, R., Padua, D., Leasure, B. and Wolfe, M. 1981. Dependence Graphs and Compiler Optimization. In *Proc. of the 8th Symp. on the Principles of Programming Languages (POPL'81), SIGPLAN Notices*. 207–218.

Larsen, L. and Harrold, M. J. 1996. Slicing object-oriented software. In *Proc. of the 18th Int'l Conf. on Software engineering (ICSE'96)*. IEEE Computer Society, 495–505.

Leuschel, M. 2002. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*. Springer LNCS 2566, 379–403.

Leuschel, M. and Sørensen, M. 1996. Redundant Argument Filtering of Logic Programs. In *Proc. of the Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR'96)*. Springer LNCS 1207, 83–103.

Leuschel, M. and Vidal, G. 2005. Forward Slicing by Conjunctive Partial Deduction and Argument Filtering. In *Proc. of the European Symposium on Programming (ESOP 2005)*. Springer LNCS 3444, 61–76.

Lloyd, J. 1994. Combining Functional and Logic Programming Languages. In *Proc. of the International Logic Programming Symposium*. 43–57.

Loogen, R., López-Fraguas, F. and Rodríguez-Artalejo, M. 1993. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93*. Springer LNCS 714, 184–200.

López-Fraguas, F. and Sánchez-Hernández, J. 1999. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*. Springer LNCS 1631, 244–247.

Moreno-Navarro, J. and Rodríguez-Artalejo, M. 1992. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming 12,* 3, 191–224.

Ochoa, C., Silva, J. and Vidal, G. 2004. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*. ACM Press, 123–134.

Reps, T. and Turnidge, T. 1996. Program Specialization via Program Slicing. In *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, O. Danvy, R. Glück, and P. Thiemann, Eds. Springer LNCS 1110, 409–429.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press

Schoenig, S. and Ducasse, M. 1996. A Backward Slicing Algorithm for Prolog. In *Proc. of the Int'l Static Analysis Symposium (SAS'96)*. Springer LNCS 1145, 317–331.

Sestoft, P. 1997. Deriving a Lazy Abstract Machine. *Journal of Functional Programming 7,* 3, 231–264.

Shapiro, E. 1983. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA.

Silva, J. and Vidal, G. 2006. Forward Slicing of Functional Logic Programs by Partial Evaluation. Computing Research Repository (CoRR) at `http://arXiv.org/abs/cs.PL/0601013/`.

Sørensen, M., Glück, R. and Jones, N. 1996. A Positive Supercompiler. *Journal of Functional Programming 6,* 6, 811–838.

Steindl, C. 1998. Intermodular Slicing of Object-oriented Programs. In *Proc. of the 7th Int'l Conf. on Compiler Construction (CC'98)*. Springer LNCS 1383, 264–279.

Terese. 2003. *Term Rewriting Systems*. Cambridge University Press.

Tip, F. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages 3*, 121–189.

Tip, F., Choi, J.-D., Field, J. and Ramalingam, G. 1996. Slicing class hierarchies in C++. *SIGPLAN Notices 31,* 10, 179–197.

Turchin, V. 1986. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems 8,* 3 (July), 292–325.

Venkatesh, G. 1991. The Semantic Approach to Program Slicing. *SIGPLAN Notices 26,* 6, 107–119. *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'91)*.

Weiser, M. 1979. Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. Ph.D. thesis, The University of Michigan.

Weiser, M. 1984. Program Slicing. *IEEE Transactions on Software Engineering 10,* 4, 352–357.

Woodward, M. R. and Allen, S. P. 1998. Slicing algebraic specifications. *Information and Software Technology 40,* 2, 105–118.

Zhao, J., Cheng, J. and Ushijima, K. 2001. A Program Dependence Model for Concurrent Logic Programs and Its Applications. In *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM'01)*. IEEE Press, 672–681.

https://doi.org/10.1017/S1471068406002870 Published online by Cambridge University Press