Forward Slicing by Conjunctive Partial Deduction and Argument Filtering[⋆]

Michael Leuschel[1] and Germán Vidal[2]

[1] School of Electronics and Computer Science, University of Southampton & Institut für Informatik, Heinrich-Heine Universität Düsseldorf
mal@ecs.soton.ac.uk
[2] DSIC, Technical University of Valencia, Camino de Vera S/N, E-46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract. Program slicing is a well-known methodology that aims at identifying the program statements that (potentially) affect the values computed at some point of interest. Within imperative programming, this technique has been successfully applied to debugging, specialization, reuse, maintenance, etc. Due to its declarative nature, adapting the slicing notions and techniques to a logic programming setting is not an easy task. In this work, we define the first, semantics-preserving, forward slicing technique for logic programs. Our approach relies on the application of a conjunctive partial deduction algorithm for a precise propagation of information between calls. We do not distinguish between static and dynamic slicing since partial deduction can naturally deal with both static and dynamic data. A slicing tool has been implemented in ECCE, where a post-processing transformation to remove redundant arguments has been added. Experiments conducted on a wide variety of programs are encouraging and demonstrate the usefulness of our approach, both as a classical slicing method and as a technique for code size reduction.

1 Introduction

Program slicing is a fundamental operation that has been successfully applied to solve many software engineering tasks, like, e.g., program understanding, maintenance, specialization, debugging, reuse, etc. Slicing was originally introduced by Weiser [32]—in the context of imperative programs—as a debugging technique. Despite its potential applications, we found very few approaches to slicing in logic programming (some notable exceptions are, e.g., [10, 27, 28, 30, 33]).

Informally, a *program slice* consists of those program statements which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *program dependence graph* [5] that makes explicit both the data and control dependences for each operation in a program. Program dependences can

⋆ This work was partially funded by the IST programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project and by the Spanish *Ministerio de Educación y Ciencia* (ref. TIN2004-00231).

be traversed backwards and forwards—from the slicing criterion—giving rise to so-called *backward* and *forward* slicing. Additionally, slices can be *static* or *dynamic*, depending on whether a concrete program's input is provided or not. More detailed information on program slicing can be found in [12, 29].

Recently, Vidal [31] introduced a novel approach to forward slicing of lazy functional logic programs. This work exploits the similarities between slicing and partial evaluation—already noticed in [25]—to compute forward slices by a slight modification of an existing partial evaluation scheme [2]. The main requirement of [31] is that the underlying partial evaluation algorithm should be—in the terminology of [26]—both *monovariant* and *monogenetic* in order to preserve the structure of the original program. Unfortunately, this requirement also restricts the precision of the computed slices.

In this work, we extend the approach of [31] in several ways. First, we adapt it to the logic programming setting. Second, we consider a polyvariant and polygenetic partial evaluation scheme: the *conjunctive* partial deduction algorithm of [3] with control based on characteristic trees [9, 18, 19]. Therefore, the computed slices are significantly more precise than those of the previous approach. Furthermore, since the basic partial deduction algorithm is kept unmodified, it can easily be implemented on top of an existing partial deduction system (in our case, ECCE [19]). Finally, we use the redundant argument filtering transformation of [21] to slice out unnecessary arguments of predicates (in addition to slicing out entire clauses).

The combination of these two approaches, [31] and [21], together with a special-purpose slicing code generator, gives rise to a simple but powerful forward slicing technique. We also pay special attention to using slicing for code size reduction. Indeed, within the ASAP project [1], we are looking at resource-aware specialization techniques, with the aim of adapting software for pervasive devices with limited resources. We hence also analyze to what extent our approach can be used as an effective code size reduction technique, to reduce the memory footprint of a program.

Our main contributions are the following. We introduce the first, semantics-preserving, forward slicing technique for logic programs that produces executable slices. While traditional approaches in the literature demand different techniques to deal with static and dynamic slicing, our scheme is general enough to produce both static and dynamic slices. In contrast to [31], the restriction to adopt a monovariant/monogenetic partial evaluation algorithm is not needed. Dropping this restriction is important as it allows us to use more powerful specialization schemes and, moreover, we do not need to modify the basic algorithm, thus easing the implementation of a slicing tool (i.e., only the code generation phase should be changed). We illustrate the usefulness of our approach on a series of benchmarks, and analyze its potential as a code-size reduction technique.

The paper is organized as follows. After introducing some foundations in the next section, Sect. 3 presents our basic approach to the computation of forward slices. Then, Sect. 4 considers the inclusion of a post-processing phase for argument filtering. Section 5 illustrates our technique by means of a detailed

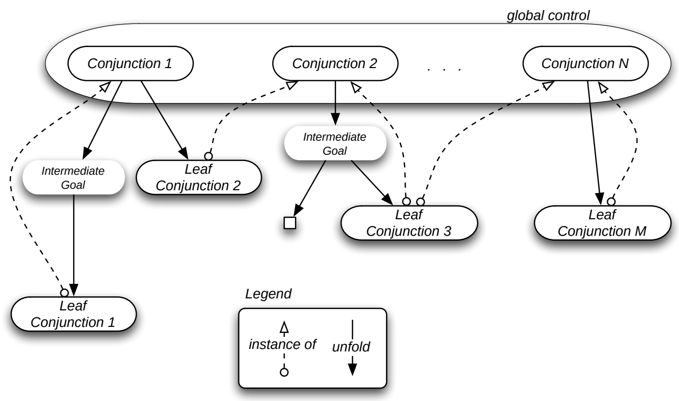

Fig. 1. The Essence of Conjunctive Partial Deduction

example, while Sect. 6 presents an extensive set of benchmarks. Finally, Sect. 7 compares some related works and concludes. More details and missing proofs can be found in [22].

2 Background

Partial evaluation [13] has been applied to many programming languages, including functional, imperative, object-oriented, logic, and functional logic programming languages. It aims at improving the overall performance of programs by pre-evaluating parts of the program that depend solely on the *static* input.

In the context of logic programming, full input to a program P consists of a goal G and evaluation corresponds to constructing a complete SLDNF-tree for $P \cup \{G\}$. For partial evaluation, the static input takes the form of a goal G' which is more general (i.e., less instantiated) than a typical goal G at runtime. In contrast to other programming languages, one can still execute P for G' and (try to) construct an SLDNF-tree for $P \cup \{G'\}$. However, since G' is not yet fully instantiated, the SLDNF-tree for $P \cup \{G'\}$ is usually infinite and ordinary evaluation will not terminate. A technique which solves this problem is known under the name of *partial deduction* [23]. Its general idea is to construct a finite number of finite, but possibly incomplete[1] SLDNF-trees and to extract from these trees a new program that allows any instance of the goal G' to be executed.

Conjunctive partial deduction (CPD) [3] is an extension of partial deduction that can achieve effects such as *deforestation* and *tupling* [24]. The essence of CPD can be seen in Fig. 1. The so-called global control of CPD generates a set $C = \{C_1, \ldots, C_n\}$ of *conjunctions* whereas the local control generates for each conjunction a possibly incomplete SLDNF-tree τ_i (a process called *unfolding*). The overall goal is to ensure that every leaf conjunction is either an instance of

[1] An SLDNF-tree is *incomplete* if, in addition to success and failure leaves, it also contains leaves where no literal has been selected for a further derivation step.

some C_i or can be split up into sub-conjunctions, each of which is an instance of some conjunction in C. This is called the *closedness* condition, and guarantees correctness of the specialized program which is then extracted by:

- generating one specialized predicate per conjunction in C (and inventing a new predicate name for it), and by producing
- one specialized clause—a *resultant*—per non-failing branch of τ_i.

A single resolution step with a specialized clause now corresponds to performing *all* the resolutions steps (using original program clauses) on the associated branch. Closedness can be ensured by various algorithms [16]. Usually, one starts off with an initial conjunction, unfolds it using some "*unfolding rule*" (a function mapping a program P and a goal G to an SLDNF-tree for $P \cup \{G\}$) and then adds all uncovered[2] leaf conjunctions to C, in turn unfolding them, and so forth. As this process is usually non-terminating, various "*generalization*" operations are applied, which, for example, can replace several conjunctions in C by a single less instantiated one. One useful foundation for the global control is based on so-called *characteristic trees*, used for example by the SP [7] and ECCE [19] specialization systems. We describe them in more detail below, as they turn out to be important for slicing.

Characteristic trees were introduced in partial deduction in order to capture all the relevant aspects of specialization. The following definitions are taken from [19] (which in turn were derived from [9] and the SP system [7]).

Definition 1 (characteristic path). *Let G_0 be a goal, and let P be a normal program whose clauses are numbered. Let $G_0, \ldots, G_n$ be the goals of a finite, possibly incomplete SLDNF-derivation D of $P \cup \{G_0\}$. The characteristic path of the derivation D is the sequence $\langle l_0 : c_0, \ldots, l_{n-1} : c_{n-1}\rangle$, where l_i is the position of the selected literal in G_i, and c_i is defined as follows:*

- *if the selected literal is an atom, then c_i is the number of the clause chosen to resolve with G_i;*
- *if the selected literal is $\neg p(\bar{t})$, then c_i is the predicate p.*

Note that an SLDNF-derivation D can be either failed, incomplete, successful, or infinite. As we will see below, characteristic paths will only be used to characterize *finite* and *nonfailing* derivations. Once the top-level goal is known, the characteristic path is sufficient to reconstruct all the intermediate goals as well as the final one.

Now that we have characterized derivations, we can characterize goals through the derivations in their associated SLDNF-trees.

Definition 2 (characteristic tree). *Let G be a goal, P a normal program, and τ a finite SLDNF-tree for $P \cup \{G\}$. Then the characteristic tree $\hat{\tau}$ of τ is the set containing the characteristic paths of the nonfailing SLDNF-derivations associated with the branches of τ. $\hat{\tau}$ is called a characteristic tree if and only if it is the characteristic tree of some finite SLDNF-tree.*

[2] I.e., those conjunctions which are not an instance of a conjunction in C.

Let U be an unfolding rule such that $U(P, G) = \tau$. Then $\hat{\tau}$ is also called the characteristic tree of G (in P) via U. We introduce the notation chtree$(G, P, U) = \hat{\tau}$. We also say that $\hat{\tau}$ is a characteristic tree of G (in P) if it is the *characteristic tree of G (in P) via some unfolding rule U.*

When characteristic trees are used to control CPD, the basic algorithm returns a set of characteristic conjunctions, $\tilde{C}$, that fulfills the conditions for the correctness of the specialization process. A *characteristic conjunction* is a pair $(C, \hat{\tau})$, where C is a conjunction of literals—a goal—and $\hat{\tau} = \mathit{chtree}(C, P, U)$ is a characteristic tree for some program P and unfolding rule U. From this set of characteristic conjunctions, the specialized program is basically obtained by unfolding and renaming.

3 Extracting Executable Forward Slices

Within imperative programming, the definition of a slicing criterion depends on whether one considers static or dynamic slicing. In the former case, a slicing criterion is traditionally defined as a pair (p, v) where p is a program statement and v is a subset of the program's variables. Then, a forward slice consists of those statements which are dependent on the slicing criterion (i.e., on the values of the variables v that appear in p), 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 [29]. As for dynamic slicing, a slicing criterion is often defined as a triple (d, i, v), where d is the input data for the program, i denotes the i-th element of the execution history, and v is a subset of the program's variables.

Adapting these notions to the setting of logic programming is not immediate. There are mainly two aspects that one should take into account:

– The execution of partially instantiated goals—thanks to the use of logic variables—makes it unclear the distinction between static and dynamic slicing.
– The lack of explicit control flow, together with the absence of side effects, makes unnecessary to consider a particular trace of the program's execution for dynamic slicing.

Therefore, we define a *slicing criterion* simply as a goal.[3] Typically, the goal will appear in the code of the source program. However, we lift this requirement for simplicity since it does affect to the forthcoming developments. A forward slice should thus contain a *subset* of the original program with those clauses that are reachable from the slicing criterion. Similarly to [27], the notion of "subset" is formalized in terms of an abstraction relation, to allow arguments to be removed, or rather replaced by a special term:

[3] If we fix an entry point to the program and restrict ourselves to a particular evaluation strategy (as in Prolog), one can still consider a concrete trace of the program. In this case, however, a standard tracer would suffice to identify the interesting goal.

Definition 3 (term abstraction). *Let $\top_t$ be the empty term (i.e., an unnamed existentially quantified variable, like the anonymous variable of Prolog). A term t is an abstraction of term t', in symbols $t \succeq t'$, iff $t = \top_t$ or $t = t'$.*

Definition 4 (literal abstraction). *An atom $p(t_1, \ldots, t_n)$ is an abstraction of atom $q(t'_1, \ldots, t'_m)$, in symbols $p(t_1, \ldots, t_n) \succeq q(t'_1, \ldots, t'_m)$, iff $p = q$, $n = m$, and $t_i \succeq t'_i$ for all $i = 1, \ldots, n$. A negative literal $\neg P$ is an abstraction of a negative literal $\neg Q$ iff $P \succeq Q$.*

Definition 5 (clause abstraction). *A clause c is an abstraction of a clause $c' = L'_0 \leftarrow L'_1, \ldots, L'_n$, in symbols $c \succeq c'$, iff $c = L_0 \leftarrow L_1, \ldots, L_n$ and $L_i \succeq L'_i$ for all $i \in \{1, \ldots, n\}$.*

Definition 6 (program abstraction). *A normal program[4] $P = (c_1, \ldots, c_n)$ is an abstraction of normal program $P' = (c'_0, \ldots, c'_m)$, in symbols $P \succeq P'$, iff $n \leq m$ and there exists a subsequence $(s_1, \ldots, s_n)$ of $(1, \ldots, m)$ such that $c_i \succeq c'_{s_i}$ for all $i \in \{1, \ldots, n\}$.*

Informally, a program P is an abstraction of program P' if it can be obtained from P' by clause deletion and by replacing some predicate arguments by the empty term $\top_t$. In the following, P is a *slice* of program P' iff $P \succeq P'$. Trivially, program slices are normal programs.

Definition 7 (correct slice). *Let P be a program and G a slicing criterion. A program P' is a correct slice of P w.r.t. G iff P' is a slice of P (i.e., $P' \succeq P$) and the following conditions hold:*
- *$P \cup \{G\}$ has an SLDNF-refutation with computed answer θ if and only if $P' \cup \{G\}$ does, and*
- *$P \cup \{G\}$ has a finitely failed SLDNF-tree if and only if $P' \cup \{G\}$ does.*

Traditional approaches to program slicing rely on the construction of some data structure which reflects the data and control dependences in a program (like, e.g., the *program dependence graphs* of [5]). The key contribution of this paper is to show that CPD can actually play such a role.

Roughly speaking, our slicing technique proceeds as follows. Firstly, given a program P and a goal G, a CPD algorithm based on characteristic trees is applied. The use of characteristic trees is relevant in our context since they record the clauses used during the unfolding of each conjunction. The complete algorithm outputs a so-called *global tree*—where each node is a characteristic conjunction—which represents an abstraction of the execution of the considered goal. In fact, this global tree contains information which is similar to that in a program dependence graph (e.g., dependences among predicate calls). In standard conjuntive partial deduction, the characteristic conjunctions, $\tilde{C}$, in the computed global tree are unfolded—following the associated characteristic trees—to produce a correct specialization of the original program (after renaming). In

[4] We consider that programs are *sequences* of clauses in order to enforce the preservation of the syntax of the original program.

order to compute a forward slice, only the code generation phase of the CPD algorithm should be changed: now, we use the characteristic tree of each conjunction in $\tilde{C}$ to determine which clauses of the original program have been used and, thus, should appear in the slice.

Given a characteristic path δ, we define $cl(\delta)$ as the set of clause numbers in this path, i.e., $cl(\delta) = \{c \mid \langle l : c\rangle$ appears in δ and c is a clause number$\}$. Program slices are then obtained from a set of characteristic trees as follows:

Definition 8 (forward slicing). *Let P be a normal program and G be a slicing criterion. Let $\tilde{C}$ be the output of the CPD algorithm (a set of characteristic conjunctions) and T be the characteristic trees in $\tilde{C}$. A forward slice of P w.r.t. G, denoted by $slice_T(P)$, contains those clauses of P that appear in some characteristic path of T. Formally, $slice_T(P) = \cup_{\hat{\tau}\in T}\{cl(\delta) \mid \delta \in \hat{\tau}\}$.*

The correctness of the forward slicing method is stated as follows:

Theorem 1. *Let P be a normal program and G be a slicing criterion. Let P' be a forward slice according to Def. 8. Then, P' is a correct slice of P w.r.t. G.*

The proof can be found in [22]. Our slicing technique produces *correct* forward slices and, moreover, is more flexible than previous approaches in the literature. In particular, in can be used to perform both dynamic and static forward slicing with a modest implementation effort, since only the code generation phase of the CPD algorithm should be changed.

4 Improving Forward Slices by Argument Filtering

The method of Def. 8 has been fully implemented in ECCE, an off-the-shelf partial evaluator for logic programs based on CPD and characteristic trees. In practice, however, we found that computed slices often contain redundant arguments that are not relevant for the execution of the slicing criterion. In order to further refine the computed slices and be able to slice out unnecessary arguments of predicates, we use the redundant argument filtering transformations (RAF) of [21].

RAF is a technique which detects certain redundant arguments (finding all redundant arguments is undecidable in general [21]). Basically, it detects those arguments which are existential and which can thus be safely removed. RAF is very useful when performed after CPD. Redundant arguments also arise when one re-uses generic predicates for more specific purposes. For instance, let us define a `member/2` predicate by re-using a generic `delete/3` predicate:

```
member(X,L) :- delete(X,L,DL).
delete(X,[X|T],T).     delete(X,[Y|T],[Y|DT]) :- delete(X,T,DT).
```

Here, the third argument of `delete` is redundant and will be removed by the partial evaluator ECCE if RAF is enabled:

```
member(X,L) :- delete(X,L).
delete(X,[X|T]).       delete(X,[Y|T]) :- delete(X,T).
```

```
int(cst(X),_,_,X).
int(var(X),Vars,Vals,R) :- lookup(X,Vars,Vals,R).
int(plus(X,Y),Vars,Vals,Res) :-
    int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX+RY.
int(minus(X,Y),Vars,Vals,Res) :-
    int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX-RY.
int(fun(X),Vars,Vals,Res) :- def0(X,Def), int(Def,Vars,Vals,Res).
int(fun(X,Arg),Vars,Vals,Res) :-
    def1(X,Var,Def), int(Arg,Vars,Vals,ResArg),
    int(Def,[Var|Vars],[ResArg|Vals],Res).
def0(one,cst(1)).
def0(rec,fun(rec)).
def1(inc,xx,plus(var(xx),cst(1))).
def1(rec,xx,fun(rec,var(xx))).
lookup(X,[X|_],[Val|_],Val).
lookup(X,[Y|T],[_|ValT],Res) :- X \= Y, lookup(X,T,ValT,Res).
```

Fig. 2. A simple functional interpreter

The ECCE system also contains the reverse argument filtering (FAR) of [21] ("reverse" because the safety conditions are reversed w.r.t. RAF). While RAF detects existential arguments (which might return a computed answer binding), FAR detects arguments which can be non-existential and non-ground but whose value is never used (and for which no computed answer binding will be returned). Consider, e.g., the following program:

```
p(X) :-  q(f(X)).     q(Z).
```

Here, the argument of `q(f(X))` is not a variable but the value is never used. The ECCE system will remove this argument if FAR is enabled:

```
p(X) :-  q.     q.
```

The elimination of redundant arguments turns out to be quite useful to remove unnecessary arguments from program slices (see next section). Only one extension is necessary in our context: while redundant arguments are deleted in [21], we replace them by the special symbol $\top_t$ so that the filtered program is still a slice—an abstraction—of the original program. The correctness of the extended slicing algorithm then follows from Theorem 1 and the results in [21].

5 Forward Slicing in Practice

In this section, we illustrate our approach to the computation of forward slices through some selected examples. Consider the program in Fig. 2 which defines an interpreter for a simple language with constants, variables, and some predefined functions. First, we consider the following slicing criterion:

```
slice1(X) :- int(minus(cst(4),plus(fun(one),cst(2))),[xx],[11],X).
```

The slice computed by ECCE w.r.t. this slicing criterion is as follows:

```
int(cst(X),_,_,X).

int(plus(X,Y),Vars,Vals,Res) :-
   int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX+RY.
int(minus(X,Y),Vars,Vals,Res) :-
   int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX-RY.
int(fun(X),Vars,Vals,Res) :- def0(X,Def), int(Def,Vars,Vals,Res).

def0(one,cst(1)).

slice1(X) :- int(minus(cst(4),plus(fun(one),cst(2))),[xx],[11],X).
```

Here, some predicates have been completely removed from the slice (e.g., `def1` or `lookup`), even though they are reachable in the predicate dependency graph. Furthermore, unused clauses are also removed, cutting down further the size of the slice. By applying the argument filtering post-processing, we get[5]

```
int(cst(X),*,*,X).

int(plus(X,Y),*,*,Res) :- int(X,*,*,RX), int(Y,*,*,RY), Res is RX+RY.
int(minus(X,Y),*,*,Res) :- int(X,*,*,RX), int(Y,*,*,RY), Res is RX-RY.
int(fun(X),*,*,Res) :- def0(X,Def), int(Def,*,*,Res).

def0(one,cst(1)).

slice1(X) :- int(minus(cst(4),plus(fun(one),cst(2))),*,*,X).
```

The resulting slice is executable and will produce the same result as the original program, e.g., the query `slice1(X)` returns the answer `X=1`. Note that this example could have been tackled by a *dynamic* slicing method, as a fully specified query was provided as the slicing criterion. It would be interesting to know how a dynamic slicer would compare against our technique, and whether we have lost any precision. In order to test this, we have implemented a simple dynamic slicer in SICStus Prolog using profiled code and extracting the used clauses using the `profile_data/4` built-in. The so extracted slice corresponds exactly to our result (without the argument filtering; see [22]), and hence no precision has been lost in this example.

In general, not only the code size of the slice is smaller but also the runtime can be improved. Thus, our forward slicing algorithm can be seen as a—rather conservative—partial evaluation method that guarantees that code size does not increase. For instance, it can be useful for resource aware specialization, when the (potential) code explosion of typical partial evaluators is unacceptable.

Our slicing tool can also be useful for program debugging. In particular, it can help the programmer to locate the source of an incorrect answer (or an

[5] For clarity, in the examples we use "`*`" to denote the empty term $\top_t$. In practice, empty terms can be replaced by any term since they play no role in the computation.

unexpected loop; finite failure is preserved in Def. 7) since it identifies the clauses that could affect the computed answer, thus easing the correction of the program. Consider, e.g., that the definition of function `plus` contains a bug:

```
int(plus(X,Y),Vars,Vals,Res) :-
    int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX-RY.
```

i.e., the programmer wrote `RX-RY` instead of `RX+RY`. Given the following goal:

```
slice2(X) :- int(plus(cst(1),cst(2)),[x],[1],X).
```

the execution returns the—incorrect—computed answer `X = -1`. By computing a forward slice w.r.t. `slice2(X)`, we get (after argument filtering) the following:

```
int(cst(X),*,*,X).
int(plus(X,Y),*,*,Res) :- int(X,*,*,RX), int(Y,*,*,RY), Res is RX-RY.
slice2(X) :- int(plus(cst(1),cst(2)),*,*,X).
```

This slice contains only 3 clauses and, thus, the user can easily detect that the definition of `plus` is wrong.

The previous two slices can be extracted by a dynamic slicing technique, since they do not involve a non-terminating goal. Now, we consider the following slicing criterion:

```
slice3(X) :- int(fun(rec),[aa,bb,cc,dd],[0,1,2,3],X).
```

Despite the fact that this goal has an infinite search space, our slicing tool returns the following slice (after argument filtering):

```
int(fun(X),*,*,*) :- def0(X,Def), int(Def,*,*,*).
def0(rec,fun(rec)).
slice3(X) :- int(fun(rec),*,*,*).
```

From this slice, the clauses which are responsible of the infinite computation can easily be identified.

6 Experimental Results

In this section, we show a summary of the experiments conducted on an extensive set of benchmarks. We used SICStus Prolog 3.11.1 (powerpc-darwin-7.2.0) and Ciao-Prolog 1.11 #221, running on a Powerbook G4, 1GHz, 1GByte of RAM. The operating system was Mac OS 10.3. We also ran some experiments with SWI Prolog 5.2.0. The runtime was obtained by special purpose benchmarker files (generated automatically be ECCE) which execute the original and specialized programs without loop overhead. The code size was obtained by using the `fcompile` command of SICStus Prolog and then measuring the size of the compiled `*.ql` files. The total speedups were obtained by the formula $\frac{n}{\sum_{i=1}^{n} \frac{spec_i}{orig_i}}$

Table 1. Speedups obtained by Specialization and by Slicing

Prolog System	SWI-Prolog		SICStus		Ciao	
Technique	Specialized	Sliced	Specialized	Sliced.	Specialized	Sliced
TOTAL	2.43	1.04	2.74	1.04	2.62	1.05
Average	5.23	1.07	6.27	1.09	11.26	1.09

where n is the number of benchmarks, and $spec_i$ and $orig_i$ are the absolute execution times of the specialized/sliced and original programs respectively.[6] The total code size reduction was obtained by the formula $1 - \frac{\sum_{i=1}^{n} specsz_i}{\sum_{i=1}^{n} origsz_i}$ where n is the number of benchmarks, and $specsz_i$ and $origsz_i$ are the code sizes of the specialized/sliced and original programs respectively.

DPPD. We first compared the slicing tool with the default conjunctive specialization of ECCE on the DPPD library of specialization benchmarks [15]. In a sense these are not typical slicing scenarios, but nonetheless give an indication of the behavior of the slicing algorithm. The experiments also allow us to evaluate to what extent our technique is useful as an alternative way to specialize programs, especially for code size reduction. Finally, the use of the DPPD library allows comparison with other reference implementations (see, e.g., [3, 14] for comparisons with MIXTUS, SP and PADDY).

Table 1 (which is a summary of the full tables in [22]) shows the speedup of the ECCE default specialization and of our slicing algorithm. Timings for SWI Prolog, SICStus Prolog, and Ciao Prolog are shown. It can be seen that the average speedup of slicing is just 4%. This shows how efficient modern Prolog implementations are, and that little overhead has to be paid for adding extra clauses 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 26.2% whereas the standard specialization has increased the code size by 56%. In the worst case, the specialization has increased the code size by 493.5% (whereas slicing never increases the code size; see the full tables in [22]).

Slicing-Specific Benchmarks. Let us now turn our attention to four, more slicing-specific experiments. Table 2 contains the results of these experiments. The `inter_medium` benchmark is the simple interpreter of Sect. 5. The `ctl_trace` benchmark is the CTL model checker from [20], extended to compute witness traces. It is sliced for a particular system and temporal logic formula to model check. The `lambdaint` benchmark is an interpreter for a simple functional language taken from [17]. It is sliced for a particular functional program (computing the Fibonacci numbers). Finally, `matlab` is an interpreter for a subset of the Matlab language (the code can be found in [22]). The overall results are very good: the code size is reduced by 60.5% and runtime decreased by 16%.

[6] Observe that this is different from the average of the speedups (which has the disadvantage that big slowdowns are not penalized sufficiently).

Table 2. Slicing Specific Benchmarks

	Slicing Time	Runtime		Size		
		Original	Sliced	Original	Sliced	Reduction
Benchmark	ms	ms	speedup	Bytes	Bytes	%
inter_medium	20	117	1.06	4798	1578	67.1%
lambdaint	390	177	1.29	7389	4769	35.5%
ctl_trace	1940	427	1.35	8053	4214	47.7%
matlab	2390	1020	1.02	27496	8303	69.8%
Total			1.16			60.5%

Table 3. Various Slicing Approaches

	Full Slicing		Simple Std. PD		Naïve PD	
Benchmark	Time (ms)	Reduction	Time (ms)	Reduction	Time (ms)	Reduction
inter_medium	20	67.1%	50	67.1%	20	41.8%
lambdaint	390	35.5%	880	9.0%	30	9.0%
ctl_trace	1940	47.7%	140	47.7%	40	1.3%
matlab	2390	69.8%	1170	69.8%	200	19.3%
Total	4740	60.5%	2240	56.4%	290	17.0%

Comparing the Influence of Local and Global Control. In Table 3, we compare the influence of the partial deduction control. Here, "Full slicing" is the standard CPD that we have used so far; "Simple Std. PD" is a standard (non-conjunctive) partial deduction with relatively simple control; and "Naïve PD" is very simple standard partial deduction in the style of [31], i.e., with a one-step unfolding and very simple generalization (although it is still more precise than [31] as it can produce some polyvariance), where we have turned the redundant argument filtering off.

The experiments we conducted (see [22] for the table of results) show the clear difference between our slicing approach and one using a naïve PD on the DPPD benchmarks used earlier: our approach manages a code size reduction of 26% whereas the naïve PD approach manages just 9.4%. The table also shows that the overall impact of the filtering is quite small. This is somewhat surprising, and may be due to the nature of the benchmarks. However, it may also mean that in the future we have to look at more powerful filtering approaches.

7 Discussion, Related and Future Work

In this work, we have introduced the first, semantics-preserving, forward slicing technique for logic programs. Traditional approaches to program slicing rely on the construction of some data structure to store the data and control dependences in a program. The key contribution of this paper has been to show that CPD can actually play such a role. The main advantages of this approach are the following: there is no need to distinguish between static and dynamic slicing and, furthermore, a slicing tool can be fully implemented with a modest implementation effort, since only the final code generation phase should be changed (i.e., the core algorithm of the partial deduction system remains untouched). A slicing tool has been fully implemented in ECCE, where a post-processing transformation to remove redundant arguments has been added. Our experiments demonstrate the usefulness of our approach, both as a classical slicing method as well as a technique for code size reduction.

As mentioned before, we are not aware of any other approach to forward slicing of logic programs. Previous approaches have only considered *backward* slicing. For instance, Schoening and Ducassé [27] defined the first backward slicing algorithm for Prolog which produces executable programs. Vasconcelos [30] introduced a flexible framework to compute both static and dynamic backward slices. Similar techniques have also been defined for constraint logic programs [28] and concurrent logic programs [33]. Within imperative programming, Field, Ramalingam, and Tip [6] introduced a *constrained* slicing scheme in which source programs are translated to an intermediate graph representation. Similarly to our approach, constrained slicing generalizes the traditional notions of static and dynamic slicing since arbitrary constraints on the input data can be made.

The closest approaches are those of [31] and [21]. Vidal [31] introduced a forward slicing method for lazy functional logic programs that exploits the similarities between slicing and partial evaluation. However, only a restrictive form of partial evaluation—i.e., monovariant and monogenetic partial evaluation—is allowed, which also restricts the precision of the computed slices. Our new approach differs from that of [31] in several aspects: we consider logic programs; we use a polyvariant and polygenetic partial evaluation scheme and, therefore, the computed slices are significantly more precise; and, moreover, since the basic partial deduction algorithm is kept unmodified, it can easily be implemented on top of an existing partial deduction system. On the other hand, Leuschel and Sørensen [21] introduced the concept of *correct erasure* in order 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. In our approach, we use this algorithm as a post-processing phase to slice out unnecessary arguments of predicates in the computed slices. The combination of these two approaches, [31] and [21], together with a special-purpose slicing code generator, form the basis of a powerful forward slicing technique.

Since our work constitutes a first step towards the development of a forward slicing technique for logic programs, there are many interesting topics for future work. For instance, an interesting topic for further research involves the computation of *backward* slices (a harder topic). In this case, the information gathered by characteristic trees is not enough and some extension is needed.

One should also investigate to what extent abstract interpretation can be used to complement our slicing technique. On its own, abstract interpretation will probably lack the precise propagation of concrete values, hence making it less suitable for dynamic slicing. However, for static slicing it may be able to remove certain clauses that a partial deduction approach cannot remove (see, e.g., [4, 8] where useless clauses are removed to complement partial deduction) and one should investigate this possibility further. One could also investigate better global control, adapted for slicing (to avoid wasted specialisation effort in case added polyvariance does not increase the precision of the slice). Finally, we can use our slicing technique as a starting point for resource aware specialization, i.e., finding a good tradeoff between code size and execution speed.

Acknowledgements. We would like to thank the anonymous referees of ESOP'05 for their valuable feedback. We also would like to thank Dan Elphick, John Gallagher, Germán Puebla, and all the other partners of the ASAP project for their considerable help and input.

References

1. Advanced Specialization and Analysis for Pervasive Computing. EU IST FET Programme Project Number IST-2001-38059. `http://www.asap.ecs.soton.ac.uk/`
2. E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 2(1):3–26, 2002.
3. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, 1999.
4. D. A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.
5. J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
6. J. Field, G. Ramalingam, and F. Tip. Parametric Program Slicing In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 379-392, ACM Press, 1995.
7. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
8. J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proc. of LOPSTR'92*, pages 151–167, Manchester, UK, 1992.
9. J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialisation. *New Generation Computing*, 9(3-4):305–333, 1991.
10. T. Gyimóthy and J. Paakki. Static Slicing of Logic Programs. In *Proc. of the 2nd Int'l Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, pages 87–103. IRISA-CNRS, 1995.
11. M. Harman and S. Danicic. Amorphous Program Slicing. In *Proc. of the 5th Int'l Workshop on Program Comprehension.* IEEE Computer Society Press, 1997.
12. M. Harman and R. Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.
13. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
14. J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 59–82. Springer-Verlag, 1996.
15. M. Leuschel. The DPPD Library of Benchmarks. Accessible from URL: `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`

16. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
17. M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specializing interpreters using offline partial deduction. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, pages 341–376. Springer LNCS 3049, 2004.
18. M. Leuschel and D. De Schreye. Constrained Partial Deduction and the Preservation of Characteristic Trees. *New Generation Computing*, 16(3):283–342, 1998.
19. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
20. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, 2000.
21. M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 83–103. Springer-Verlag, 1996.
22. M. Leuschel and G. Vidal. Forward Slicing by Conjunctive Partial Deduction and Argument Filtering. Technical Report, DSSE, University of Southamtpon, December 2004.
23. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
24. A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *The Journal of Logic Programming*, 19,20:261–320, 1994.
25. T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle*, pages 409–429. Springer LNCS 1110, 1996.
26. R. Glück and M.H. Sørensen A Roadmap to Metacomputation by Supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle*, pages 137–160. Springer LNCS 1110, 1996.
27. S. Schoenig and M. Ducassé. A Backward Slicing Algorithm for Prolog. In *Proc. of the Int'l Static Analysis Symposium (SAS'96)*, pages 317–331. Springer LNCS 1145, 1996.
28. G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.
29. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
30. W. Vasconcelos. A Flexible Framework for Dynamic and Static Slicing of Logic Programs. In *Proc. of the First Int'l Workshop on Practical Aspects of Declarative Languages (PADL'99)*, pages 259–274. Springer LNCS 1551, 1999.
31. G. Vidal. Forward slicing of multi-paradigm declarative programs based on partial evaluation. In M. Leuschel, editor, *Proc. of Logic-based Program Synthesis and Transformation (LOPSTR'02)*, LNCS 2664, pages 219–237. Springer-Verlag, 2003.
32. M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
33. J. Zhao, J. Cheng, and K. Ushijima. A Program Dependence Model for Concurrent Logic Programs and Its Applications. In *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM'01)*, pages 672–681. IEEE Press, 2001.