Università di Pisa

Dipartimento di Informatica

Technical Report : TR-96-20

A Program Specialiser for Meta-level Compositions of Logic Programs

Antonio Brogi, Simone Contiero

May 24, 1996

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

A Program Specialiser for Meta-level Compositions of Logic Programs

Antonio Brogi and Simone Contiero

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
{brogi,contiero}@di.unipi.it

Abstract

Meta-level compositions of object logic programs are naturally implemented by means of meta-programming techniques. Meta-interpreters defining program compositions however suffer from a computational overhead that is due partly to the interpretation layer present in all meta-programs, and partly to the specific interpretation layer needed to deal with program compositions.

We show that meta-interpreters implementing compositions of object programs can be fruitfully specialised w.r.t. meta-level queries of the form `Demo(E,G)`, where `E` denotes a program expression and `G` denotes a (partially instantiated) object level query. More precisely, we describe the design and implementation of a declarative program specialiser that suitably transforms such meta-interpreters so as to sensibly reduce — if not to completely remove — the overhead due to the handling of program compositions. In many cases the specialiser succeeds in eliminating also the overhead due to meta-interpretation.

Contents

ACM Computing Classification

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic — *logic programming*; I.2.2 [**Artificial Intelligence**]: Automatic Programming — *program transformation*

General Terms: Languages, Theory

Additional Key Words and Phrases: Logic programs, composition operations, meta-logic, program transformation

1 Introduction

A simple, yet powerful, extension of logic programming consists of introducing a set of meta-level operations for composing separate object programs. The introduction of program composition operations enhances the expressive power and knowledge representation capabilities of logic programming by supporting a wealth of applications (e.g., see [4, 6]) ranging from software engineering applications, such as modular programming, to artificial intelligence applications, such as hierarchical and hypothetical reasoning.

Meta-level compositions of object programs are naturally implemented by meta-programming. As shown in [6, 8], several operations over definite logic programs can be implemented by extending the so-called *vanilla* meta-interpreter [33]. In this paper, we will consider an extended vanilla meta-interpreter $\mathtt{M}_\pi$ that supports a number of program composition operations over a collection π of object programs. In particular, we will refer to an existing implementation of the meta-interpreter in the system `COMPOSE` [5], a logic programming system written in the programming language Gödel [20].

The meta-interpreter $\mathtt{M}_\pi$ can be exploited to prove meta-level queries of the form `Demo(E,G)`, where `E` denotes a composition of object programs and `G` is an object level query. The meta-interpreter suffers from a computational overhead which is due partly to the interpretation layer present in all meta-programs, and partly to the specific interpretation layer needed to deal with program compositions.

The main objective of this paper is to eliminate, or at least to reduce, such an overhead. It is well-known that partial evaluation [13] (also referred to as partial deduction [21] in logic programming) can be exploited to specialise programs so as to improve their efficiency. In particular, the application of partial evaluation techniques for reducing the interpretation overhead in meta-programs has been studied by many authors (e.g., [14, 21, 29, 35]).

We will describe the design, implementation and application of a program specialiser which has been constructed according to the following requirements.

(1) The specialiser should be targeted to improving the efficiency of the meta-interpreter $\mathtt{M}_\pi$. Rather than aiming at designing a general specialiser for logic programs, our objective is to build an *ad hoc* specialiser to be suitably integrated with the system `COMPOSE` [5].

(2) The specialiser should be written in a *declarative* programming style. As pointed out in [3, 19, 20], dynamic meta-programming applications (such as program specialisation) require the adoption of a so-called *ground* representation of object programs in order to encompass declarativeness.

(3) Finally, we would like to prove the correctness of the program specialiser by applying, as much as possible, general results from the theory of partial evaluation in logic programming [26]. Rather than extending the basic results of partial evaluation to the extension considered, as done for instance in [10], we would like to reduce our application to a special instance of the general case of partial evaluation in logic programming.

Several program specialisers for logic programs have been developed (e.g., [16, 21, 32, 35]). The majority of them is written in Prolog and employs extra-logical features of the languages, such as the cut operator and built-in predicates like `var`, `nonvar`, `assert`, `retract`, and `copy_term`. While most of these specialisers yield notable speed-ups, the adoption of Prolog's extra-logical features spoils declarativeness, which remains one of the distinguishing features of logic programming [3, 20].

As mentioned before, our objective is to build a program specialiser targeted to improving the efficiency of the meta-interpreter $\mathtt{M}_\pi$. Since the currently available implementation of $\mathtt{M}_\pi$ in the system `COMPOSE` is written in Gödel, the first obvious solution is to try to apply `SAGE` [17], an existing partial evaluator for Gödel programs. Even if `SAGE` is in principle applicable to any Gödel program, it has been primarily designed for a special class of meta-programs, which does not include our meta-interpreter $\mathtt{M}_\pi$. Unfortunately, as we will discuss thoroughly in Sect. 5, `SAGE` does not offer a substantial speed-up when specialising $\mathtt{M}_\pi$. This led us to implement in Gödel a specialiser targeted for the extended meta-interpreter $\mathtt{M}_\pi$. Besides directly ensuring the integration with `COMPOSE`, this choice was also suggested by the wide support offered by Gödel for developing meta-programming applications using the ground representation of object programs.

The behaviour of the specialiser can be partitioned into two phases: A partial evaluation phase and a program transformation phase. In the first phase, given the extended vanilla meta-interpreter $\mathtt{M}_\pi$ (containing the representation of a set π of object programs) and a query of the form `Demo(E,G)`, a new program $\mathtt{M}_\pi^s$ specialised w.r.t. queries of the form `Demo(E,G)` is produced by applying standard partial evaluation techniques based on unfolding. The two requirements $\mathtt{M}_\pi^s$ must satisfy are that:

(1) $\texttt{Demo(E,G)}$ should have the same answers for $\texttt{M}_\pi^s$ and for $\texttt{M}_\pi$, and

(2) $\texttt{Demo(E,G)}$ should run more efficiently on $\texttt{M}_\pi^s$ than on $\texttt{M}_\pi$.

More precisely, given a program expression `E`, we will address the problem of partially evaluating the meta-interpreter $\texttt{M}_\pi$ w.r.t. queries of the form `Demo(E,x)`, where the object level query `x` is not specified. There are two main reasons for this choice. First, when using the meta-interpreter $\texttt{M}_\pi$, the same program expression `E` will be in general employed to query many different object goals. It is therefore convenient to obtain a partially evaluated meta-interpreter that can be used to query *any* object level goal on a program expression `E`. Moreover, general existing techniques can be employed to further specialise the obtained meta-interpreter w.r.t. partially instantiated object queries `G`.

The correctness of the partial evaluation performed by the program specialiser is established by proving the equivalence between the meta-interpreter $\texttt{M}_\pi$ and its partially evaluated version $\texttt{M}_\pi^s$. The proof suitably resorts to general results from the theory of partial evaluation in logic programming [26].

After the first partial evaluation phase, the specialiser further optimises the obtained meta-program $\texttt{M}_\pi^s$ by applying program transformation techniques (without further input information) thus obtaining a new program $\texttt{New\_M}_\pi$. Two alternative program transformation techniques are applied to $\texttt{M}_\pi^s$, depending on the structure of the given program expression `E`. In one case, the program transformation eliminates from $\texttt{M}_\pi$ both the overhead due to the interpretation layer and the overhead due to the specific interpretation layer needed to deal with program compositions. In the other case, a simple transformation (based on introducing new predicates) is applied, which still sensibly reduces the second type of overhead. The equivalence between $\texttt{M}_\pi^s$ and $\texttt{New\_M}_\pi$ is proved in both cases.

As we will discuss extensively in Sect. 5, the program specialiser yields a notable performance speed-up, and does not generate an explosion of the program code, as it may happen with uncontrolled unfolding.

Plan of the paper
In Sect. 2 we introduce the extended vanilla meta-interpreter $\texttt{M}_\pi$ defining a family of program composition operations. In Sect. 3 we recall some background results from the theory of partial evaluation in logic programming. Sect. 4 is devoted to present the design and implementation

of our program specialiser. We first establish a general equivalence result showing when partial evaluation techniques can be applied to the extended vanilla meta-interpreter so as to preserve the semantics of the initial program. Then we show how a concrete partial evaluator can be built in order to achieve these results. Finally, we describe the optimisation techniques applicable during a second program transformation phase, together with suitable equivalence results. Sect. 5 contains an assessment of the overall approach, and a comparison between our approach and `SAGE`. Finally, Sect. 6 contains a discussion of related work along with some concluding remarks and directions for future work.

2 Meta-level compositions of object programs

Four basic operations for composing definite logic programs are introduced: Union (denoted by `Union`), intersection (`Inters`), encapsulation (`Enc`), and import (`Import`). These operations have been thoroughly described in [4, 6, 8], where their semantics and implementation have been studied.

We recall here the implementation of program composition operations by means of meta-programming, as described in [5]. Let us first introduce a simple extension of the *vanilla* meta-interpreter [33], where a two-argument predicate `Demo` is used to denote provability. (In the following we will adhere to the syntax conventions used in Gödel [20]: Identifiers of variables start with lower case letters, while all other identifiers start with upper case letters.)

```
(1) Demo(e,Empty).
(2) Demo(e,(y And z)) <- Demo(e,y) & Demo(e,z).
(3) Demo(e,y) <- Clause(e,y If z) & Demo(e,z).
```

The standard vanilla meta-interpreter is used to interpret a single object program `e`, whose clauses are represented at the meta-level by means of the predicate `Clause`. Namely, `Demo(e,y)` states that the formula `y` is provable in the program `e`. The procedural reading of the vanilla-meta-interpreter is the following. The first unit clause states that the `Empty` goal is solved in any object program. The second clause deals with conjunctive goals: To solve a conjunction `y And z`, solve both `y` and `z`. Finally, the third clause deals with the general case of goal reduction. To solve an atomic goal `y`, choose a clause from the program `e` whose head unifies with the goal, and recursively solve its body.

The vanilla meta-interpreter can be employed to interpret compositions of object programs by extending the definition of Clause to represent program compositions, and by introducing a predicate Statement to represent (collections of) object programs. Each program composition operation is represented at the meta-level by a functor whose meaning is defined by means of the Clause predicate:

```
(4) Clause(e1 Union e2, z If w ) <-  Clause(e1, z If w ).
(5) Clause(e1 Union e2, z If w ) <-  Clause(e2, z If w ).
(6) Clause(e1 Inters e2, z If (w And u)) <-  Clause(e1, z If w) &
                                             Clause(e2, z If u).
(7) Clause(Enc(e), z If Empty) <-  Demo(e,z).
(8) Clause(e1 Import e2, z If w) <-  Clause(e1, z If u) &
                                     Partition(u, w, z) &
                                     Demo(e2, z).
(9) Clause(Pr(p), z If w) <-  Statement(p, z If w).
```

Clauses (4) and (5) define the Union operation. Namely, a clause belongs to the meta-level representation of the composition e1 Union e2 if it belongs either to the meta-level representation of e1 or to the meta-level representation of e2. Clause (6) states that a clause z If (w And u) belongs to the meta-level representation of e1 Inters e2 if there are two clauses, one in e1 and the other in e2, such that their heads unify in z and the conjunction of their bodies, suitably instantiated, equals (w And u). The meta-level representation of an encapsulated program Enc(e) — clause (7) — consists of unit clauses of the form Clause(Enc(e), z If Empty) where z is provable in e. In this way, the code of e is hidden to other programs, which may only refer to the set of sentences provable in e. Clause (8) defines the Import operation that builds a module e1 Import e2 out of a pair of modules e1 and e2, which play the role of the public and private part of the module, respectively. The clauses in e1 Import e2 are obtained from the clauses of e1 by dropping calls to hidden predicates in the original clause body, provided that they are provable in the private part e2 (and possibly instantiating the public calls). The "library" predicate Partition, whose definition is omitted for brevity, simply partitions a given set (i.e., a conjunction) of atoms u into two sets (i.e., conjunctions) w and z. Via backtracking, Partition returns all the possible ways (w and z) in which u can be split. Finally, clause (9) states that the definition of the predicate Clause resorts to the definition of the predicate Statement when its first argument, Pr(p), denotes a single object program.

Object level programs are represented by means of the `Statement` predicate. For instance, program `Plus`:

```
Plus(Zero,x,x).
Plus(S(x),y,S(z)) <- Plus(x,y,z).
```

is represented at the meta-level by the clauses:

```
Statement(Plus, Plus(Zero,x,x) If Empty).
Statement(Plus, Plus(S(x),y,S(z)) If Plus(x,y,z)).
```

The extended vanilla meta-interpreter described so far employs the *non-ground* representation of object programs, in which object level variables are denoted by variables at the meta-level. An alternative, equivalent formulation of the extended meta-interpreter using the ground representation of object programs is described in [5]. A thorough discussion of the relative merits of the two representations is outside the scope of this paper and can be found in [5, 18, 19, 27]. However, since the extended vanilla meta-interpreter does not modify the object programs, the use of a non-ground representation of object programs is preferable since the resulting meta-interpreter has a natural declarative semantics [27] besides the usual simplicity and efficiency. In this paper, we will therefore consider the extended vanilla meta-interpreter using the non-ground representation, as defined by clauses (1)—(9) above.

The operations `Union`, `Inters`, `Enc`, and `Import` define the following class `Exp` of program expressions `Exp`:

```
Exp ::= Exp Union Exp | Exp Inters Exp | Enc(Exp) | Exp Import Exp | Pr(P)
```

where `P` is (the name of) a definite object program. From now onwards, we will refer to the elements of `Exp` by `E`, `F` (possibly subscripted). Moreover, we will denote by $\mathtt{M}_\pi$ the meta-interpreter consisting of clauses (1)—(9), together with the meta-level representation of a set π of object programs.

From a programming point of view, the basic set of composition operations considered (union, intersection, encapsulation, import) enhances the expressive power and knowledge representation capabilities of logic programming. Many different applications of (meta-)programming by composing separate object programs have been illustrated. Software engineering applications, where existing software components are to be combined to build complex systems, are one class of examples. The application of program composition operations to modular programming is extensively discussed in [6], where various program structuring policies are analysed, as well as forms of information hiding and import/export between modules. Program composition operations have also been employed for the rational reconstruction of typical artificial intelligence applications, such as hierarchical reasoning, hypothetical reasoning, and default reasoning. The interested reader may refer to [4], for a variety of examples of usage of the operators for programming.

We conclude this section by describing a simple example of use of the meta-level operations for programming.

Example 1 Consider a graph whose nodes represent cities and whose edges represent train connections between cities, labelled by the type of train (`Reg`, `Nat` or `Int`). Suppose that paths between cities are defined by a program `Path`, whose meta-level representation is:

```
Statement(Path, Path(city1,city2) If Train(city1,city2,type)).
Statement(Path, Path(city1,city2) If Train(city1,city3,type) And Path(city3,city2)).
```

Let `Train` be a database containing information on existing trains and represented at the meta-level by the following set of unit clauses:

```
Statement(Train, Train(Florence,Pisa,Reg) If Empty).
Statement(Train, Train(Florence,Rome,Int) If Empty).
Statement(Train, Train(Pisa,Genova,Nat) If Empty).
Statement(Train, Train(Pisa,Rome,Nat) If Empty).
Statement(Train, Train(Milan,Florence,Int) If Empty).
Statement(Train, Train(Milan,Pisa,Nat) If Empty).
```

Then a query of the form

```
Demo(Pr(Path) Union Pr(Train), Path(Florence,x))
```

can be employed to find the cities reachable from Florence. For instance, in the meta-program $\mathtt{M}_{\pi}$, where $\pi = \{$ `Train`, `Path` $\}$, the following refutation establishes the existence of a train path between Florence and Pisa.

```
    <-   Demo(Pr(Train) Union Pr(Path), Path(Florence,Pisa))
{ applying clause (3) }
    <-   Clause(Pr(Train) Union Pr(Path), Path(Florence,Pisa) If w) &
         Demo(Pr(Train) Union Pr(Path),w)
{ applying clause (5) }
    <-   Clause(Pr(Path), Path(Florence,Pisa) If w) &
         Demo(Pr(Train) Union Pr(Path),w)
{ applying clause (9) }
    <-   Statement(Path, Path(Florence,Pisa) If w) &
         Demo(Pr(Train) Union Pr(Path),w)
{ applying the first clause for Statement with substitution (w = Train(Florence,Pisa,type))}
    <-   Demo(Pr(Train) Union Pr(Path), Train(Florence,Pisa,type) )
{ applying clause (3) }
    <-   Clause(Pr(Train) Union Pr(Path), Train(Florence,Pisa,type) If x) &
         Demo(Pr(Train) Union Pr(Path),x)
{ applying clause (4) }
    <-   Clause(Pr(Train), Train(Florence,Pisa,type) If x) &
         Demo(Pr(Train) Union Pr(Path),x)
{ applying clause (9) }
    <-   Statement(Train, Train(Florence,Pisa,type) If x) &
         Demo(Pr(Train) Union Pr(Path),x)
{ applying the third clause for Statement with substitution (type = Reg, x = Empty) }
    <-   Demo(Pr(Train) Union Pr(Path), Empty)
{ applying clause (1) }
    □
```

Program composition operations can be exploited to build more structured queries. For instance, one may be interested in finding paths between cities in which each intermediate connection is supported by `Int` trains. Then a program expression of the form

```
Pr(Path) Union (Pr(Train) Inters Pr(IC))
```

can be employed, where the program `IC` defines the constraint on the kind of train connections to be considered for constructing the path. `IC` can be represented at the meta-level by the statement:

```
Statement(IC, Train(x,y,Int) If Empty).
```

□

3 Background: Partial evaluation in logic programming

This section is devoted to recall some basic definitions and theoretical foundations of partial evaluation in logic programming, taken from [26]. The interested reader may also refer to [18, 30] for an introduction to partial evaluation and program transformation in logic programming.

Let us first introduce the notion of *resultant* that is needed in the definition of partial evaluation.

Definition 1 *A* resultant *is a first order formula of the form $Q_1 \leftarrow Q_2$, where Q_i is either absent or a conjunction of literals ($i = 1, 2$). Any variables in Q_1 or Q_2 are assumed to be universally quantified at the front of the resultant.*

Definition 2 *Let P be a (normal) program, G a (normal) goal $\leftarrow Q$ and $G_0 = G, G_1, \ldots G_n$ a SLD(NF)-derivation of $P \cup \{G\}$, where the sequence of substitutions is $\vartheta_1, \ldots, \vartheta_n$ and G_n is $\leftarrow Q_n$. Let ϑ be the restriction of $\vartheta_1 \ldots \vartheta_n$ to the variables in G. Then we say the derivation has* length n *with* computed answer ϑ *and* resultant $Q\vartheta \leftarrow Q_n$. *(If $n = 0$, the resultant is $Q \leftarrow Q$.)*

Notice that a resultant is not in general a clause, because the Q on the left stands for a conjunction of literals. A resultant encapsulates most of the information about the result of a derivation. Moreover, a resultant is particularly significant when the original goal G consists of a single atom $\leftarrow A$. In this case the resultant is the program clause $A\vartheta \leftarrow Q_n$ and the original derivation from the goal $\leftarrow A$ is equivalent to (i.e., has the same resultant as) a single SLD-resolution step with this new clause [26].

We now recall the formal definitions of partial evaluation in logic programming.

Definition 3 *Let P be a program, A an atom, and T a $SLD(NF)$ tree for $P \cup \{\leftarrow A\}$. Let $G_1, \ldots, G_r$ be non-root goals in T chosen so that each non-failing branch in T contains exactly one of them. Let R_i ($i = 1, \ldots, r$) be the resultant of the derivation from $\leftarrow A$ down to G_i, given by the branch leading to G_i. Then the set of clauses $R_1, \ldots, R_r$ is called a* partial evaluation *of A in P.*

If $\mathbf{A} = \{A_1, \ldots, A_k\}$ is a finite set of atoms, then a *partial evaluation of* $\mathbf{A}$ *in P is the union of partial evaluations of $A_1, \ldots, A_k$ in P.*

Definition 4 *A* partial evaluation *of P w.r.t. a finite set of atoms $\mathbf{A}$ is a program obtained from P by replacing the set of clauses of P whose head contains one of the predicate symbols appearing in $\mathbf{A}$ (called the* partially evaluated predicates*) with a partial evaluation of $\mathbf{A}$ in P.*

Moreover, a partial evaluation of P w.r.t. $\mathbf{A}$ *using SLD-trees* is a partial evaluation in which all the SLDNF-trees used for the partial evaluation are actually SLD-trees.

Two general conditions, called *closedness* and *independence*, are introduced in [26] to establish the equivalence between the original program and its partial evaluations.

Definition 5 *Let S be a set of first order formulae and $\mathbf{A}$ a finite set of atoms. We say that S is* $\mathbf{A}$-closed *if each atom in S containing a predicate symbol occurring in an atom in $\mathbf{A}$ is an instance of an atom in $\mathbf{A}$.*

Definition 6 *Let $\mathbf{A}$ be a finite set of atoms. We say $\mathbf{A}$ is* independent *if no pair of atoms in $\mathbf{A}$ have a common instance.*

We finally recall one of the main equivalence results presented in [26].

Proposition 1 *Let P be a (normal) program, G a (normal) goal, $\mathbf{A}$ a finite, independent set of atoms, and P' a partial evaluation of P w.r.t. $\mathbf{A}$ using SLD-trees such that $P' \cup \{G\}$ is $\mathbf{A}$-closed. Then the following hold:*

(i) *$P' \cup \{G\}$ has a SLD-refutation with computed answer ϑ iff $P \cup \{G\}$ does, and*

(ii) *$P' \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.*

Example 2 Consider the simple program `P`:

```
Q(y) <- R(y).
R(x).
```

A partial evaluation of `P` w.r.t. $\mathbf{A} = \{$`Q(B)`$\}$ is the program `P1`:

```
Q(B) <- R(B).
R(x).
```

Proposition 1 ensures that, if `G=`$\leftarrow$`Q(B)` or `G=`$\leftarrow$`R(z)`, then `P` and `P1` are equivalent (since `P1` $\cup$ {`G`} is **A**-closed). If, instead, `G=`$\leftarrow$`Q(A)` then `P1` $\cup${`G`} is not **A**-closed and we have that `P` $\cup${$\leftarrow$ `Q(A)`} succeeds, while `P1` $\cup${$\leftarrow$ `Q(A)`} fails. □

According to the proposition 1, from now onwards we will consider two programs equivalent if and only if they have the same computed answers and finite failures w.r.t. a set of given goals.

We end this section recalling that, as illustrated in [2, 26], the **A**-closedness condition can be replaced by a slightly weaker (and slightly more complicated) condition, called **A**-coveredness. However, since in our setting the two conditions collapse, we will keep on referring to the **A**-closedness condition.

4 A program specialiser for meta-level compositions of programs

We now present the design and implementation of a program specialiser for the extended vanilla meta-interpreter $\mathtt{M}_\pi$ introduced in Sect. 2. We first describe the partial evaluation phase, and then the program transformation phase. The actual implementation of the whole specialiser is discussed at the end of the section. An assessment of the program specialiser will be discussed in Sect. 5.

4.1 Phase I: Partial evaluation

Given the extended vanilla meta-interpreter $\mathtt{M}_\pi$ and a program expression `E`, the partial evaluation phase should yield a new, specialised meta-program $\mathtt{M}_\pi^s$ that is equivalent to $\mathtt{M}_\pi$ w.r.t. queries of the form `Demo(E,G)`, for any object query `G`. We will indeed show how the general results described in Sect. 3 can be suitably applied to establish such an equivalence result.

As pointed out for instance in [2, 24], it is crucial to develop some strategy to control unfolding. For instance, in our case, the unfolding of predicate `Demo` hardly ever increases the "degree of specialisation" of the meta-interpreter $\mathtt{M}_\pi$ w.r.t. a query of the form `Demo(E,x)`. Indeed, since the second argument of `Demo` in the initial query is not specified, the clauses defining `Demo` can be only applied "blindly". Therefore, unfolding `Demo` generally increases the size of the code of the meta-interpreter rather than its degree of specialisation. While successful derivations may be shorter, the size of the whole SLD-tree corresponding to the unfolded program is generally larger, and this vanishes the former advantage, wiping out any efficiency gain. Moreover, an uncontrolled unfolding of Demo does not terminate since clause (2) in the definition of Demo can be applied for ever.

For instance, a partial evaluation of $\mathtt{M}_\pi$ w.r.t. Demo(E,x), obtained by unfolding predicate Demo and considering the first two levels of the SLD-tree for $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Demo(E,x)}\}$, generates the following result:

```
Demo(E,Empty).
Demo(E,(Empty And x)) <- Demo(E,x).
Demo(E,(x1 And x2 And x3)) <- Demo(E,x1) & Demo(E,x2) & Demo(E,x3).
Demo(E,(x1 And x2)) <- Clause(E,x1 If y1) & Demo(E,y1) & Demo(E,x2).
Demo(E,x) <- Clause(E,x If Empty).
Demo(E,x) <- Clause(E,x If y1 And y2) & Demo(E,y1) & Demo(E,y2).
Demo(E,x) <- Clause(E,x If y) & Clause(E,y If z) & Demo(E,z).
```

In order to specialise $\mathtt{M}_\pi$ w.r.t. queries of the form Demo(E,x), we will then partially evaluate $\mathtt{M}_\pi$ w.r.t. predicate Clause. Indeed, the very structure of $\mathtt{M}_\pi$ shows that the definition of Demo relies on the definition of Clause to perform each reduction step. Actually, the extra interpretation layer needed to deal with program compositions relies on predicate Clause. We will show that, since the program expressions we consider are fully specified, the unfolding of Clause is an effective way of specialising $\mathtt{M}_\pi$.

4.1.1 Equivalence result

In order to show when a partial evaluation of the meta-interpreter $\mathtt{M}_\pi$obtained by unfolding the predicate Clause is equivalent to $\mathtt{M}_\pi$, it is necessary to determine the set of atoms $\mathbf{A}$ w.r.t. which the meta-interpreter $\mathtt{M}_\pi$ will be partially evaluated. Since Clause is defined recursively on the structure of program expressions, the definition of Clause obtained by unfolding $\mathtt{M}_\pi$ w.r.t. Clause(E,x) may contain, in some clause body, atoms of the form Clause(F,y), where F is a sub-expression of E. By definition of partial evaluation (def. 4), Clause(F,y) does not unify with any statement in the specialised program $\mathtt{M}'_\pi$. Hence $\mathtt{M}'_\pi$ will not be in general equivalent to $\mathtt{M}_\pi$, since $\mathtt{M}'_\pi$ will fail on queries on which $\mathtt{M}_\pi$ succeeds, as illustrated by the following example.

Example 3 Consider the programs P1 and P2 represented at the meta-level by the clauses:

```
(s1)  Statement(P1, A If B).
(s2)  Statement(P2, B If Empty).
```

Consider the program expression `Pr(P1) Union Pr(P2)`. Then a partial evaluation of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}$ = {`Clause(Pr(P1) Union Pr(P2),x)`} is:

```
(1)  Demo(e,Empty).
(2)  Demo(e,(y And z)) <- Demo(e,y) & Demo(e,z).
(3)  Demo(e,y) <- Clause(e,y If z) & Demo(e,z).
(u1) Clause(Pr(P1) Union Pr(P2), z If w) <- Clause(Pr(P1), z If w).
(u2) Clause(Pr(P1) Union Pr(P2), z If w) <- Clause(Pr(P2), z If w).
(s1) Statement(P1, A If B).
(s2) Statement(P2, B If Empty).
```

It is easy to see that the query `Demo(Pr(P1) Union Pr(P2),A)` succeeds in the meta-interpreter $\mathtt{M}_\pi$ (consisting of clauses `(1)` — `(9)`,`(s1)`,`(s2)`), while it cannot be refuted by the specialised meta-interpreter $\mathtt{M}'_\pi$ (consisting of clauses `(1)`,`(2)`,`(3)`,`(u1)`,`(u2)`,`(s1)`,`(s2)`). □

In order to specialise $\mathtt{M}_\pi$ w.r.t. a query `Demo(E,x)`, we will therefore consider a partial evaluation of $\mathtt{M}_\pi$ w.r.t. the set of atoms $\mathbf{A}=\{\mathtt{Clause(E,x)}, \mathtt{Clause(E_1,x)}, \ldots, \mathtt{Clause(E_n,x)}\}$ where $\{\mathtt{E}, \mathtt{E_1}, \ldots, \mathtt{E_n}\}$ is the set of all sub-expressions of `E`. The set $\Gamma^*(\mathtt{E})$ of all sub-expressions of a program expression `E` can be defined inductively as follows.

Definition 7 *Let* `E`, `F` *be program expressions. Then:*

$$
\begin{aligned}
\Gamma^*(\mathtt{E\ Union\ F}) &= \{\mathtt{E\ Union\ F}\} \cup \Gamma^*(\mathtt{E}) \cup \Gamma^*(\mathtt{F}) \\
\Gamma^*(\mathtt{E\ Inters\ F}) &= \{\mathtt{E\ Inters\ F}\} \cup \Gamma^*(\mathtt{E}) \cup \Gamma^*(\mathtt{F}) \\
\Gamma^*(\mathtt{Enc(E)}) &= \{\mathtt{Enc(E)}\} \cup \Gamma^*(\mathtt{E}) \\
\Gamma^*(\mathtt{E\ Import\ F}) &= \{\mathtt{E\ Import\ F}\} \cup \Gamma^*(\mathtt{E}) \cup \Gamma^*(\mathtt{F}) \\
\Gamma^*(\mathtt{Pr(P)}) &= \{\mathtt{Pr(P)}\}
\end{aligned}
$$

We can now state the main equivalence result we are interested in.

Proposition 2 *Let* `E` *be a program expression, and let* $\mathbf{A}=\{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$. *Let* $\mathtt{M}'_\pi$ *be a partial evaluation of* $\mathtt{M}_\pi$ *w.r.t.* $\mathbf{A}$. *Then, for any object level query* `G`, *the following hold:*

(i) $\mathtt{M}'_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *has a SLD-refutation with computed answer* ϑ *iff* $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *does, and*

(ii) $\mathtt{M}'_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *has a finitely failed SLD-tree iff* $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *does.*

Unfortunately, proposition 2 is not a direct corollary of proposition 1 (Sect. 3). Indeed the specialised program $\mathtt{M}'_\pi$ in proposition 2 does not satisfy the closedness condition w.r.t. the set $\mathbf{A}=\{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$. Namely, the definition of `Demo` in $\mathtt{M}'_\pi$ (viz., the same as in $\mathtt{M}_\pi$) contains the clause

```
(3) Demo(e,y) <- Clause(e,y If z) & Demo(e,z).
```

where `Clause(e, y If z)` is not an instance of any atom in $\mathbf{A}$.

One way of directly applying proposition 1 could be to consider the partial evaluation of $\mathtt{M}_\pi$ w.r.t. the above set of atoms $\mathbf{A}$ extended with the set of atoms $\{\mathtt{Demo(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$. This would satisfy the closedness condition at the cost of generating a larger code for $\mathtt{M}'_\pi$. Namely $\mathtt{M}'_\pi$ would contain at least three clauses for each atom of the form `Demo(F,x)` in $\mathbf{A}$. This solution is not quite convenient since the new clauses for `Demo` increase the size of the code of the meta-interpreter without really increasing its degree of specialisation and thus negatively affect the performance of the program.

We will instead show that proposition 2 holds, and that its proof can be obtained by suitably exploiting proposition 1. Namely, in order to prove the equivalence between $\mathtt{M}_\pi$ and $\mathtt{M}'_\pi$ of proposition 2, we will consider a third program $\mathtt{M}''_\pi$ which can be proved equivalent — by virtue of proposition 1 — both to $\mathtt{M}_\pi$ and to $\mathtt{M}'_\pi$. Hence this, by transitivity of equivalence, establishes that proposition 2 holds.

Given a program expression `E`, we consider the two sets of atoms $\mathbf{A}_1 = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$, and $\mathbf{A}_2 = \{\mathtt{Demo(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$. The proof of proposition 2 is then partitioned in three parts (see Figure 1 — where $Pe(M,\mathbf{A})$ denotes a partial evaluation of program M w.r.t. $\mathbf{A}$).

(i) We first show (Lemma 1) that every partial evaluation of $\mathtt{M}_\pi$ performed in two stages, first w.r.t. $\mathbf{A}_1$, and then w.r.t. $\mathbf{A}_2$, can be equivalently performed in a single step w.r.t. the set $\mathbf{A}_1\cup\ \mathbf{A}_2$. Namely, we show that every program $\mathtt{M}''_\pi$ obtained by first partially evaluating $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1$ yielding $\mathtt{M}'_\pi$, and then by partially evaluating $\mathtt{M}'_\pi$ w.r.t. $\mathbf{A}_2$, can be also obtained by partially evaluating $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1 \cup \mathbf{A}_2$.

(ii) We then show (Lemma 2) that every partial evaluation $\mathtt{M}''_\pi$ of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1 \cup \mathbf{A}_2$ is equivalent to $\mathtt{M}_\pi$ as far as queries of the form `Demo(E,G)` are concerned.

(iii) Finally we show (Lemma 3) that, given a partial evaluation $\mathtt{M}'_\pi$ of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1$, every partial evaluation $\mathtt{M}''_\pi$ of $\mathtt{M}'_\pi$ w.r.t. $\mathbf{A}_2$ is equivalent to $\mathtt{M}'_\pi$ as far as queries of the form `Demo(E,G)` are concerned.

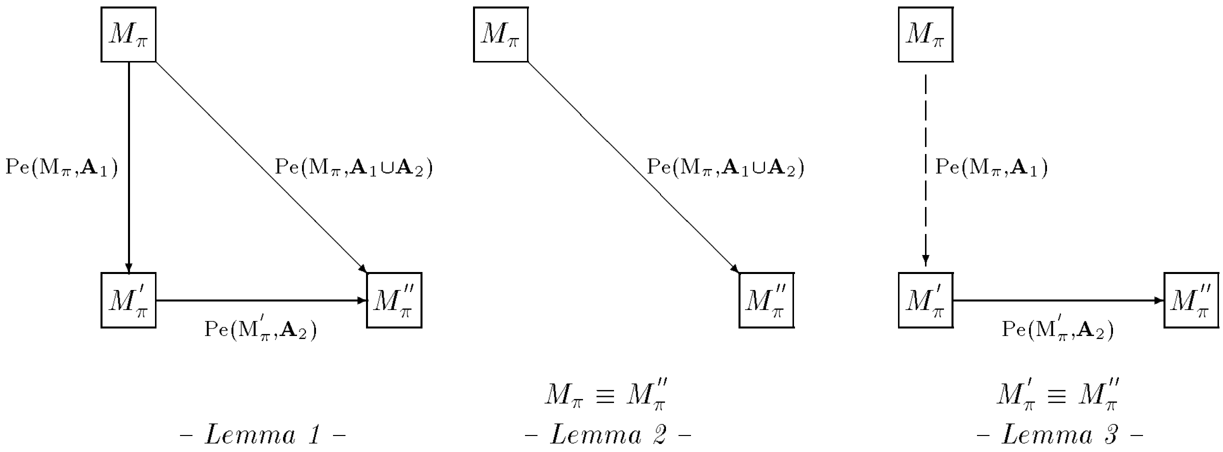

Figure 1: Structure of the proof of Proposition 2.

Lemma 1 *Let* `E` *be a program expression, and consider the two sets* $\mathbf{A}_1 = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$, *and* $\mathbf{A}_2 = \{\mathtt{Demo(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$. *Let* $\mathtt{M}'_\pi$ *be a partial evaluation of* $\mathtt{M}_\pi$ *w.r.t.* $\mathbf{A}_1$, *and let* $\mathtt{M}''_\pi$ *be a partial evaluation of* $\mathtt{M}'_\pi$ *w.r.t.* $\mathbf{A}_2$. *Then* $\mathtt{M}''_\pi$ *is a partial evaluation of* $\mathtt{M}_\pi$ *w.r.t.* $\mathbf{A}_1 \cup \mathbf{A}_2$.

Proof.
Consider a program $\mathtt{M}''_\pi$ that is a partial evaluation of $\mathtt{M}'_\pi$ w.r.t. $\mathbf{A}_2$, where $\mathtt{M}'_\pi$ is in turn a partial evaluation of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1$. By definition of partial evaluation, $\mathtt{M}''_\pi$ is a finite set of clauses defining the predicates `Demo`, `Clause`, `Statement` (and `Partition`). Since `Statement` (and `Partition`) are not among the partially evaluated predicates (see definition 4), their definition in $\mathtt{M}''_\pi$ is the same as in $\mathtt{M}_\pi$, and is therefore the same as in any partial evaluation of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1 \cup\ \mathbf{A}_2$.
The definition of `Clause` in $\mathtt{M}''_\pi$ is extracted from the SLD trees for $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Clause(F,x)}\}$, for each

$\mathtt{F} \in \Gamma^*(\mathtt{E})$. Obviously the same definition for `Clause` can be obtained as a partial evaluation of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1\cup\ \mathbf{A}_2$, by considering the same SLD trees and resolvents.
The definition of `Demo` in $\mathtt{M}_\pi''$ is extracted from the SLD trees for $\mathtt{M}_\pi'\cup\{\leftarrow \mathtt{Demo(F,x)}\}$, for each $\mathtt{F} \in \Gamma^*(\mathtt{E})$. The same definition can be extracted from the SLD trees for the goals $\leftarrow \mathtt{Demo(F,x)}$, with $\mathtt{F} \in \Gamma^*(\mathtt{E})$, and the program $\mathtt{M}_\pi$ rather than $\mathtt{M}_\pi'$. Indeed for each derivation $d = G_0G_1G_2\ldots$ for $G_0 = \leftarrow \mathtt{Demo(F,x)}$ in $\mathtt{M}_\pi'$ there exists a corresponding derivation $d' = G_0\Delta_1\Delta_2\ldots$ for $G_0 = \leftarrow \mathtt{Demo(F,x)}$ in $\mathtt{M}_\pi$, where each Δ_i is a (finite) sequence of goals $H_1^iH_2^i\ldots H_{n_i}^iG_i$. More precisely, each Δ_{i+1} is defined as follows:
(1) If G_{i+1} is obtained from G_i by applying a statement for a predicate other than `Clause`, then $\Delta_{i+1} = G_{i+1}$, that is, the same statement is applied in the derivations d and d'.
(2) Suppose that G_{i+1} is instead obtained from G_i by applying a statement $A \leftarrow \overline{B}$ defining `Clause` in $\mathtt{M}_\pi'$. By definition of $\mathtt{M}_\pi'$, $A \leftarrow \overline{B}$ is a resultant of some SLD-derivation $K_0K_1\ldots K_m$ for $\mathtt{M}_\pi\cup\{K_0\}$ where $K_0 = \leftarrow \mathtt{Clause(F',x)}$ for some $\mathtt{F'} \in \Gamma^*(\mathtt{E})$ and where $K_1\ldots K_m$ are obtained by applying a sequence of statements $C_1\ldots C_m$ (suitably instantiated) from $\mathtt{M}_\pi$. Then Δ_{i+1} is the sequence of goals $H_1H_2\ldots H_mG_{i+1}$ obtained by applying the sequence of statements $C_1\ldots C_m$ (suitably instantiated) as in the derivation for $\mathtt{M}_\pi\cup\{K_0\}$. Namely, each H_{j+1} is obtained from H_j by applying clause C_i to the atom corresponding to the atom selected in K_i. $\square$

We now prove that every partial evaluation $\mathtt{M}_\pi''$ of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1\cup\ \mathbf{A}_2$ is equivalent to $\mathtt{M}_\pi$ as far as queries of the form `Demo(E,G)` are concerned. We first observe a property of SLD derivations for $\mathtt{M}_\pi$, which follows immediately from the definition of the meta-interpreter $\mathtt{M}_\pi$.

Observation 1 *Let* `E` *be a program expression, and let* $\leftarrow G$ *be a goal of the form* $\leftarrow \mathtt{Demo(E,x)}$ *or* $\leftarrow \mathtt{Clause(E,x)}$. *Let* $G\vartheta \leftarrow G'$ *be a resultant corresponding to a SLD derivation for* $\mathtt{M}_\pi \cup\{\leftarrow G\}$. *Then* $(\mathtt{Demo(F,t)} \in G' \ \vee\ \mathtt{Clause(F,t)} \in G') \Longrightarrow \mathtt{F} \in \Gamma^*(\mathtt{E})$.

Lemma 2 *Let* `E` *be a program expression, and let* $\mathbf{A} = \mathbf{A}_1\cup\ \mathbf{A}_2$, *where* $\mathbf{A}_1 = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$ *and* $\mathbf{A}_2 = \{\mathtt{Demo(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$. *Let* $\mathtt{M}_\pi''$ *be a partial evaluation of* $\mathtt{M}_\pi$ *w.r.t.* $\mathbf{A}$. *Then the following hold:*

(i) $\mathtt{M}_\pi''\cup\{\leftarrow \mathtt{Demo(E,G)}\}$ *has a SLD-refutation with computed answer* ϑ *iff* $\mathtt{M}_\pi\cup\{\leftarrow \mathtt{Demo(E,G)}\}$ *does,*

(ii) $\mathtt{M}_\pi''\cup\{\leftarrow \mathtt{Demo(E,G)}\}$ *has a finitely failed SLD-tree iff* $\mathtt{M}_\pi\cup\{\leftarrow \mathtt{Demo(E,G)}\}$ *does.*

Proof.

The proof descends from proposition 1, since the following conditions hold.

(1) $\mathbf{A}$ *is a finite set.* Immediate, by the inductive definition of $\Gamma^*(\mathtt{E})$.

(2) $\mathbf{A}$ *is an independent set*, that is, no pair of atoms in $\mathbf{A}$ have a common instance. This immediately follows from the fact that $\Gamma^*(\mathtt{E})$ is a set of ground terms. Therefore $\{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$ is an independent set as well as $\{\mathtt{Demo(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$. Hence, since $\mathtt{Clause} \neq \mathtt{Demo}$, we have that $\mathbf{A}$ is an independent set.

(3) $\mathtt{M}_\pi^{''} \cup \{\mathtt{Demo(E,G)}\}$ *is* $\mathbf{A}$*-closed.* We have to show that any atom in $\mathtt{M}_\pi^{''} \cup \{\mathtt{Demo(E,G)}\}$ containing a predicate symbol occurring in an atom in $\mathbf{A}$ is an instance of an atom in $\mathbf{A}$. We first observe that the query `Demo(E,G)` is an instance of the atom $\mathtt{Demo(E,x)} \in \mathbf{A}$. We then analyse the clauses of $\mathtt{M}_\pi^{''}$. The head of each clause defining predicate `Clause` or `Demo` is an instance of an atom in $\mathbf{A}$, by definition of partial evaluation. Moreover, observation 1 ensures that every atom of the form `Clause(F,t)` or `Demo(F,t)` occurring in some clause body of $\mathtt{M}_\pi^{''}$ is an instance of an atom in $\mathbf{A}$, since `F` belongs to $\Gamma^*(\mathtt{E})$. Finally, the definition of predicate `Statement` (made of unit clauses only), and the definition of predicate `Partition` do not contain any atom with a predicate symbol occurring in an atom in $\mathbf{A}$. □

We now prove that, given a partial evaluation $\mathtt{M}_\pi^{'}$ of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1 = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$, every partial evaluation $\mathtt{M}_\pi^{''}$ of $\mathtt{M}_\pi^{'}$ w.r.t. $\mathbf{A}_2 = \{\mathtt{Demo(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$ is equivalent to $\mathtt{M}_\pi^{'}$ as far as queries of the form `Demo(E,G)` are concerned.

Lemma 3 *Let* `E` *be a program expression, let* $\mathbf{A}_1 = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$ *and let* $\mathbf{A_2} = \{\mathtt{Demo(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$. *Let* $\mathtt{M}_\pi^{'}$ *be a partial evaluation of* $\mathtt{M}_\pi$ *w.r.t.* $\mathbf{A}_1$*, and let* $\mathtt{M}_\pi^{''}$ *be a partial evaluation of* $\mathtt{M}_\pi^{'}$ *w.r.t.* $\mathbf{A}_2$*. Then the following hold:*

- *(i)* $\mathtt{M}_\pi^{'} \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *has a SLD-refutation with computed answer* ϑ *iff* $\mathtt{M}_\pi^{''} \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *does,*
- *(ii)* $\mathtt{M}_\pi^{'} \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *has a finitely failed SLD-tree iff* $\mathtt{M}_\pi^{''} \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *does.*

Proof.

As for lemma 2, the proof descends from proposition 1.

(1) $\mathbf{A}_2$ *is a finite set.* Immediate, by the inductive definition of $\Gamma^*(\mathtt{E})$.

(2) $\mathbf{A}_2$ *is an independent set.* Immediate, since in part (2) of the proof of lemma 2 we proved that $\mathbf{A}_1 \cup \mathbf{A}_2$ is an independent set. A fortiori $\mathbf{A}_2$ is an independent set.

(3) $\mathtt{M}''_\pi \cup \{\mathtt{Demo(E,G)}\}$ *is* $\mathbf{A}_2$*-closed.* Analogous to part (3) of the proof of lemma 2. □

We are finally in the position to prove proposition 2.

Proof of Proposition 2.
The proof descends from lemmas 1, 2 and 3. Indeed we have shown (lemma 1) that every partial evaluation of $\mathtt{M}_\pi$ performed in two stages, first w.r.t. $\mathbf{A}_1$, and then w.r.t. $\mathbf{A}_2$, can be equivalently performed in a single step w.r.t. the set $\mathbf{A}_1 \cup \mathbf{A}_2$. Then we have shown (lemma 2) that every partial evaluation $\mathtt{M}''_\pi$ of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1 \cup \mathbf{A}_2$ is equivalent to $\mathtt{M}_\pi$ as far as queries of the form `Demo(E,G)` are concerned. Finally we have shown (lemma 3) that, given a partial evaluation $\mathtt{M}'_\pi$ of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}_1$, every partial evaluation $\mathtt{M}''_\pi$ of $\mathtt{M}'_\pi$ w.r.t. $\mathbf{A}_2$ is equivalent to $\mathtt{M}'_\pi$ as far as queries of the form `Demo(E,G)` are concerned. Hence, by the transitivity of program equivalence, we have that $\mathtt{M}_\pi$ and $\mathtt{M}'_\pi$ are equivalent. □

4.1.2 Partial evaluation strategy

In the previous section we have stated (proposition 2) that *any* partial evaluation of the meta-interpreter $\mathtt{M}_\pi$ w.r.t. the set of atoms $\mathbf{A} = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$ is equivalent to $\mathtt{M}_\pi$ as far as queries of the form `Demo(E,G)` are concerned. Proposition 2 however establishes a general result without giving any clue neither on *which* partial evaluation to choose nor on *how* to actually build it. We will now focus on the first of these issues, while the second one will be discussed in Sect. 4.3. According to definition 3 (Sect. 3), the profile of a partial evaluation of a program w.r.t. an atom depends on two choices:

(i) Which *computation rule* is employed in the construction of the SLD tree (the computation rule is implicit in definition 3), and

(ii) Which set of nodes $G_1, \ldots, G_r$ is chosen to build the set of clauses defining the specialised program.

In order to design our partial evaluation strategy, we have therefore to fix these two choices.

As far as the computation rule is concerned, we consider any computation rule that "gives priority" to predicates `Clause` and `Statement`. Simply stated, for each goal G, we consider rules that always select an atom with predicate `Clause` or `Statement`, if any. If there is no such atom in G, then no constraint is imposed and the rule is free to choose any atom in G.

Definition 8 *A computation rule $\mathcal{R}$* gives priority *to a set of predicates Σ if for all goals G:*

$$(\mathcal{R}(G) = A \ \wedge \ Pred(A) \notin \Sigma) \implies (\forall B \in G: \ Pred(B) \notin \Sigma)$$

where $\mathcal{R}(G)$ denotes the atom in G selected by R, and $Pred(A)$ denotes the predicate symbol of atom A. A computation rule $\mathcal{R}$ that gives priority to a set of predicates Σ will be denoted by $\mathcal{R}_\Sigma$.

In order to partially evaluate the meta-interpreter $\mathtt{M}_\pi$, we therefore consider computation rules that give priority to the set of predicates $\mathtt{CS} = \{\mathtt{Clause}, \mathtt{Statement}\}$. Intuitively speaking, a computation rule $\mathcal{R}_{\mathtt{CS}}$ mirrors the idea of first unfolding the "promising" atoms (i.e., those which can be specialised by unfolding), while postponing the unfolding of all other atoms.

Let us now turn our attention to which nodes to select in the SLD tree in order to build the set of resultants (viz., clauses) defining a partial evaluation of $\mathtt{M}_\pi$. Namely, we specify the way in which the goals $G_1, \ldots, G_r$ in definition 3 are chosen.

Let T be a SLD-tree for $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Clause}(\mathtt{F}, \mathtt{x})\}$, built using a computation rule $\mathcal{R}_{\mathtt{CS}}$. The set of nodes $G_1, \ldots, G_r$ is chosen as the set of all (non-root) nodes in T such that for each G_j:

($\mathtt{S}_1$) For each proper ancestor N of G_j: $Pred(\mathcal{R}_{\mathtt{CS}}(N)) \in \mathtt{CS}$, and

($\mathtt{S}_2$) G_j is either a success leaf of T, or a non-leaf node of T such that $Pred(\mathcal{R}_{\mathtt{CS}}(G_j)) \notin \mathtt{CS}$.

We have therefore defined a particular strategy of partial evaluation for the meta-interpreter $\mathtt{M}_\pi$, which consists of:

(i) Adopting any computation rule $\mathcal{R}_{\mathtt{CS}}$ in the construction of SLD trees, and

(ii) Selecting the nodes in the SLD trees according to rules ($\mathtt{S}_1$) and ($\mathtt{S}_2$).

In the sequel, we will call any partial evaluation following such a strategy a ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)-*partial evaluation.* Indeed we have to show that every ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)-partial evaluation of $\mathtt{M}_\pi$ w.r.t.
$\mathbf{A} = \{\mathtt{Clause}(\mathtt{F}, \mathtt{x}) \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$ (where $\mathtt{E}$ is a program expression) *is* a partial evaluation, that is, it matches with definition 3. In order to prove this (Proposition 3 below) we first observe a property of SLD-derivations in $\mathtt{M}_\pi$.

Lemma 4 *Let* $\mathtt{F}$ *be a program expression. Let* d *be any SLD-derivation for* $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Clause(F,x)}\}$ *in which only atoms with predicate* `Clause` *or* `Statement` *are selected for clause reduction. Then* d *has finite length.*

Proof.

The proof is by structural induction on `F`.

(*Base case*) Suppose that `F=Pr(P)`. Then the goal $\leftarrow \mathtt{Clause(Pr(P),x)}$ can be rewritten only by means of clause (9) of $\mathtt{M}_\pi$ yielding the new goal $\leftarrow \mathtt{Statement(P,x)}$. Since the definition of predicate `Statement` consists of units clauses only, then any derivation of the goal $\leftarrow \mathtt{Clause(Pr(P),x)}$ in $\mathtt{M}_\pi$ has finite length.

(*Inductive case*) Suppose that `F=F1 Union F2`. Then the goal $\leftarrow$ `Clause(F1 Union F2,x)` can be rewritten only by clause (4) or (5) of $\mathtt{M}_\pi$ yielding the new goal $\leftarrow \mathtt{Clause(F1,x)}$ or $\leftarrow \mathtt{Clause(F2,x)}$. By inductive hypothesis, any derivation in $\mathtt{M}_\pi$ of these two goals has finite length, and hence any derivation of the goal $\leftarrow$ `Clause(F1 Union F2,x)` in $\mathtt{M}_\pi$ has finite length. The case `F=F1 Inters F2` is analogous. Suppose that `F=Enc(F1)`. Then the goal $\leftarrow$ `Clause(Enc(F1),x)` can be rewritten only by clause (7) of $\mathtt{M}_\pi$ yielding the new goal $\leftarrow \mathtt{Demo(F1,x)}$. Since, by hypothesis, only atoms with predicate `Clause` or `Statement` can be selected, the derivation terminates here. The case of `F=F1 Import F2` is analogous. □

Proposition 3 *Let* `E` *be a program expression, let* $\mathtt{F} \in \Gamma^*(\mathtt{E})$*, and let* T *be a SLD tree for* $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Clause(F,x)}\}$*, built using a computation rule* $\mathcal{R}_{\mathtt{CS}}$*. Let* $G_1, \ldots, G_r$ *be non-root goals in* T *chosen according to rules* $(\mathtt{S}_1)$ *and* $(\mathtt{S}_2)$*. Let* R_i *(*$i = 1, \ldots, r$*) be the resultant of the derivation from* $\leftarrow \mathtt{Clause(F,x)}$ *down to* G_i*, given by the branch leading to* G_i*. Then the set of clauses* $R_1, \ldots, R_r$ *is a partial evaluation of* `Clause(F,x)` *in* $\mathtt{M}_\pi$*.*

Proof.

By referring to definition 3, we only have to show that each non-failing branch in T contains *exactly one* of $G_1, \ldots, G_r$.

We first show that each non-failing branch in T contains *at least* one of $G_1, \ldots, G_r$. Let us first consider successful branches in T. Let $d = \gamma_1\gamma_2\ldots\gamma_n$ be a successful derivation in T, where $\gamma_1 = \leftarrow \mathtt{Clause(F,x)}$, and γ_n is the empty goal. Now, if $\forall i \in \{1, \ldots, n-1\} : \mathcal{R}_{\mathtt{CS}}(\gamma_i) \in \mathtt{CS}$ then, by rule $(\mathtt{S}_1)$, $\exists j \in \{1, \ldots, r\} : G_j = \gamma_n$. Suppose instead that $\exists i \in \{1, \ldots, n-1\} : \mathcal{R}_{\mathtt{CS}}(\gamma_i) \notin \mathtt{CS}$. Let then $m = min\{i \mid \mathcal{R}_{\mathtt{CS}}(\gamma_i) \notin \mathtt{CS}\}$. Then, by rules $(\mathtt{S}_1)$ and $(\mathtt{S}_2)$, $\exists j \in \{1, \ldots, r\} : G_j = \gamma_m$.

Let us now consider infinite branches in T. Let $d = \gamma_1\gamma_2\ldots$ an infinite derivation in T where

$\gamma_1 = \leftarrow \mathtt{Clause(F,x)}$. Since d has infinite length then, by lemma 4, $\exists i : \mathcal{R}_{\mathtt{CS}}(\gamma_i) \notin \mathtt{CS}$. Again, let $m = min\{i \mid \mathcal{R}_{\mathtt{CS}}(\gamma_i) \notin \mathtt{CS}\}$. Then, by rules $(\mathtt{S}_1)$ and $(\mathtt{S}_2)$, $\exists j \in \{1, \ldots, r\} : G_j = \gamma_m$.
We now show that each non-failing branch in T contains *at most* one of $G_1, \ldots, G_r$. Suppose that $\exists G_i, G_j \in \{G_1, \ldots, G_r\} : G_i$ is a proper ancestor of G_j in T. Since G_i is a non-leaf node then, by rule $(\mathtt{S}_2)$, $Pred(\mathcal{R}_{\mathtt{CS}}(G_i)) \notin \mathtt{CS}$. On the other hand, rule $(\mathtt{S}_1)$ states that $Pred(\mathcal{R}_{\mathtt{CS}}(G_i)) \in \mathtt{CS}$ since G_i is a proper ancestor of G_j. Contradiction. □

Since a ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)-partial evaluation of $\mathtt{M}_\pi$ is a particular case of partial evaluation according to definition 3, then the equivalence result of proposition 2 carries over to ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)-partial evaluations.

Corollary 1 *Let* $\mathtt{E}$ *be a program expression, let* $\mathbf{A} = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$ *and let* $\mathtt{M}'_\pi$ *be a* ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)*-partial evaluation of* $\mathtt{M}_\pi$ *w.r.t.* $\mathbf{A}$*. Then, for any object level query* $\mathtt{G}$*, the following hold:*

(i) $\mathtt{M}'_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *has a SLD-refutation with computed answer* ϑ *iff* $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *does,*

(ii) $\mathtt{M}'_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *has a finitely failed SLD-tree iff* $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *does.*

Proof.
Immediate by propositions 2 and 3. □

It is worth observing that the notion of ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)-partial evaluation allows us to shrink the set of atoms $\mathbf{A} = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in \Gamma^*(\mathtt{E})\}$ w.r.t. which the partial evaluation of the meta-interpreter $\mathtt{M}_\pi$ is performed. Indeed, for any program expression `E`, it is sufficient to consider the set of atoms of the form `Clause(F,x)` such that `F` is either `E` itself or some "critical" sub-expression of `E`. A critical sub-expression $\mathtt{F}_i$ of `E` is a sub-expression of `E` that is either encapsulated ($\mathtt{Enc(F}_i\mathtt{)}$) or imported by some other expression ($\mathtt{F}_j$ `Import` $\mathtt{F}_i$). Indeed, these cases correspond to the cases in which the definition of predicate `Clause` resorts to predicate `Demo` in the meta-interpreter $\mathtt{M}_\pi$.

The set $\Gamma(\mathtt{E})$ of "critical" sub-expression of `E` can be inductively defined as follows.

Definition 9 *Let* `E`*,* `F` *be program expressions. Then:*

$$\Gamma(\mathtt{E\ Union\ F}) = \Gamma(\mathtt{E}) \cup \Gamma(\mathtt{F})$$

$$
\begin{aligned}
\Gamma(\texttt{E Inters F}) &= \Gamma(\texttt{E}) \cup \Gamma(\texttt{F}) \\
\Gamma(\texttt{Enc(E)}) &= \{\texttt{E}\} \cup \Gamma(\texttt{E}) \\
\Gamma(\texttt{E Import F}) &= \{\texttt{F}\} \cup \Gamma(\texttt{E}) \cup \Gamma(\texttt{F}) \\
\Gamma(\texttt{Pr(P)}) &= \{\}
\end{aligned}
$$

It is easy to observe that, for each program expression E we have that $(\Gamma(\texttt{E}) \cup \{\texttt{E}\}) \subseteq \Gamma^*(\texttt{E})$, and hence that $\mathbf{A}' = \{\texttt{Clause(F,x)} \mid \texttt{F} \in (\Gamma(\texttt{E}) \cup \{\texttt{E}\})\} \subseteq \mathbf{A} = \{\texttt{Clause(F,x)} \mid \texttt{F} \in \Gamma^*(\texttt{E})\}$.

Obviously, shrinking the set of atoms on which to perform the partial evaluation directly reduces the size of the code of the specialised program.

We now prove that the definition of $(R_{\texttt{CS}},\texttt{S}_1,\texttt{S}_2)$-partial evaluation indeed allows us to partially evaluate the meta-interpreter $\texttt{M}_\pi$ w.r.t. the reduced set of atoms $\mathbf{A}' = \{\texttt{Clause(F,x)} \mid \texttt{F} \in (\Gamma(\texttt{E}) \cup \{\texttt{E}\})\}$, still preserving the equivalence results established in proposition 2 and corollary 1.

Proposition 4 *Let* E *be a program expression, and let* $\mathbf{A}' = \{\texttt{Clause(F,x)} \mid \texttt{F} \in (\Gamma(\texttt{E}) \cup \{\texttt{E}\})\}$. *Let* $\texttt{M}^s_\pi$ *be a* $(R_{\texttt{CS}},\texttt{S}_1,\texttt{S}_2)$*-partial evaluation of* $\texttt{M}_\pi$ *w.r.t.* $\mathbf{A}'$*. Then, for any object level query* G*, the following hold:*

(i) $\texttt{M}^s_\pi \cup \{\leftarrow \texttt{Demo(E,G)}\}$ *has a SLD-refutation with computed answer* ϑ *iff* $\texttt{M}_\pi \cup \{\leftarrow \texttt{Demo(E,G)}\}$ *does,*

(ii) $\texttt{M}^s_\pi \cup \{\leftarrow \texttt{Demo(E,G)}\}$ *has a finitely failed SLD-tree iff* $\texttt{M}_\pi \cup \{\leftarrow \texttt{Demo(E,G)}\}$ *does.*

Proof.
We first show that any $(R_{\texttt{CS}},\texttt{S}_1,\texttt{S}_2)$-partial evaluation $\texttt{M}^s_\pi$ of $\texttt{M}_\pi$ w.r.t. $\mathbf{A}'$ is equivalent to a $(R_{\texttt{CS}},\texttt{S}_1,\texttt{S}_2)$-partial evaluation $\texttt{M}'_\pi$ of $\texttt{M}_\pi$ w.r.t. $\mathbf{A} = \{\texttt{Clause(F,x)} \mid \texttt{F} \in \Gamma^*(\texttt{E})\}$, as far as queries of the form Demo(E,G) are concerned.
Since $\mathbf{A}' \subseteq \mathbf{A}$, a $(R_{\texttt{CS}},\texttt{S}_1,\texttt{S}_2)$-partial evaluation $\texttt{M}'_\pi$ of $\texttt{M}_\pi$ w.r.t. $\mathbf{A}$ can be viewed as a $(R_{\texttt{CS}},\texttt{S}_1,\texttt{S}_2)$-partial evaluation $\texttt{M}^s_\pi$ of $\texttt{M}_\pi$ w.r.t. $\mathbf{A}'$, extended with a set of clauses Δ corresponding to the partial evaluation of the set of atoms $(\mathbf{A} - \mathbf{A}')$.
Intuitively speaking, the equivalence of $\texttt{M}'_\pi$ and $\texttt{M}^s_\pi$ is due to the fact that no derivation for $\leftarrow$ $\texttt{Demo(E,G)}$ in $\texttt{M}'_\pi$ ever uses clauses in Δ. To show this we first notice that for any $\texttt{F} \in (\Gamma(\texttt{E}) \cup \{\texttt{E}\})$ the relation

$(\texttt{Clause(F,t)} \leftarrow \overline{B} \in \texttt{M}'_\pi) \implies (\forall B_i \in \overline{B} : \mathit{Pred}(B_i) = \texttt{Partition} \vee (B_i = \texttt{Demo(F}',\texttt{t}') \wedge \texttt{F}' \in \Gamma(\texttt{E})))$ holds, by definition of ($R_{\texttt{CS}}$,$\texttt{S}_1$,$\texttt{S}_2$)-partial evaluation and by definition of Γ. By virtue of this relation, if $d = \gamma_1\gamma_2\ldots$ is a (possibly infinite) derivation for $\leftarrow \texttt{Demo(E,G)}$ in $\texttt{M}'_\pi$ (where $\gamma_1 = \leftarrow \texttt{Demo(E,G)}$), then for any atom of the form Demo(F,t) or Clause(F,t), in any resolvent γ_i, the condition $\texttt{F} \in \Gamma(\texttt{E})$ holds.
Therefore, since all clauses in Δ have the form $\texttt{Clause(F,x)} \leftarrow B_1\&\ldots\&B_k$ with $\texttt{F} \in \Gamma^*(\texttt{E}) - (\Gamma(\texttt{E}) \cup \{\texttt{E}\})$, no derivation for $\leftarrow \texttt{Demo(E,G)}$ in $\texttt{M}^s_\pi$ uses clauses in Δ. Hence $\texttt{M}^s_\pi$ and $\texttt{M}'_\pi$ are equivalent as far as queries of the form Demo(E,G) are concerned.
Finally we observe that, by corollary 1, if $\texttt{M}'_\pi$ is a ($R_{\texttt{CS}}$,$\texttt{S}_1$,$\texttt{S}_2$)-partial evaluation of $\texttt{M}_\pi$ w.r.t. the set $\mathbf{A} = \{\texttt{Clause(F,x)} \mid \texttt{F} \in \Gamma^*(\texttt{E})\}$, then $\texttt{M}_\pi$ and $\texttt{M}'_\pi$ are equivalent w.r.t. queries of the form Demo(E,G) for any object level query G. This concludes our proof. □

4.1.3 Independence on the computation rule

As pointed out in [26], it is interesting to analyse the role of computation rules in the equivalence results concerning partial evaluations of logic programs.

Proposition 2 has been proved for all computation rules used in obtaining the partial evaluation $\texttt{M}'_\pi$ of $\texttt{M}_\pi$. As far as the equivalence of $\texttt{M}'_\pi$ and $\texttt{M}_\pi$ w.r.t. goals is concerned, Proposition 2 states that, for instance, "if $\texttt{M}'_\pi \cup \{\leftarrow \texttt{Demo(E,G)}\}$ has a refutation with computed answer ϑ using some computation rule, then $\texttt{M}_\pi \cup \{\leftarrow \texttt{Demo(E,G)}\}$ has a refutation with computed answer ϑ, which may use another, possibly different computation rule." Indeed, since $\texttt{M}_\pi$ and $\texttt{M}'_\pi$ are both definite programs, from the independence of the computation rule theorem [25], part (i) of proposition 2 (concerning refutations) can be stated for any three computation rules, one to be used in the partial evaluation, another for $\texttt{M}'_\pi$, and a third for $\texttt{M}_\pi$. In particular, this means that the same computation rule can be used throughout. Notice that, as pointed out in [26], the same independence result does not hold for failures, that is, for part (ii) of proposition 2. Namely a goal may have, using the same computation rule, a finitely failed SLD-tree in $\texttt{M}'_\pi$ and an infinite derivation in $\texttt{M}_\pi$. However, as observed in [26], part (ii) of proposition 2 holds for any *fair* computation rule, by the completeness result of negation as failure for definite programs [25].

Similar considerations apply to corollary 1 and proposition 4. Namely, the equivalence between $\texttt{M}_\pi$ and its partial evaluation holds for any pair of fair computation rules to be used for the two programs. Differently from proposition 2, however, corollary 1 and proposition 4 have been proved for

$\mathcal{R}_{\mathtt{CS}}$ computation rules (i.e., computation rules giving priority to predicates `Clause` and `Statement`) used in obtaining the partial evaluation of $\mathtt{M}_\pi$.

4.2 Phase II: Program transformation

In the previous section we have shown how partial evaluation techniques can be exploited to specialise the meta-interpreter $\mathtt{M}_\pi$ (introduced in Sect. 2) w.r.t. queries of the form `Demo(E,x)`, for some given program expression `E`. More precisely, we have shown that any ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)-partial evaluation $\mathtt{M}^s_\pi$ of $\mathtt{M}_\pi$ w.r.t. the set of atoms $\mathbf{A}' = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in (\Gamma(\mathtt{E}) \cup \{\mathtt{E}\})\}$ is equivalent to $\mathtt{M}_\pi$ (proposition 4). In this section we describe how program transformation techniques can be applied to further optimise the meta-interpreter $\mathtt{M}^s_\pi$ obtained at the end of the partial evaluation phase.

A first optimisation step consists of deleting from $\mathtt{M}^s_\pi$ the clauses defining predicate `Statement`. Indeed predicate `Statement` is *totally* evaluated during the partial evaluation phase. Namely the definition of predicate `Clause` in $\mathtt{M}^s_\pi$ does not depend on the definition of `Statement` (as it does in $\mathtt{M}_\pi$). Therefore, the definition of `Statement` can be removed altogether from $\mathtt{M}^s_\pi$ since `Statement` is isolated in the dependency graph of $\mathtt{M}^s_\pi$. (Similarly, the definition of predicate `Partition` can be eliminated from $\mathtt{M}^s_\pi$ whenever the program expression `E` — w.r.t. which $\mathtt{M}_\pi$ was partially evaluated — does not contain `Import` operations.)

It is worth observing that the elimination of predicate `Statement` does not affect the equivalence between $\mathtt{M}_\pi$ and $\mathtt{M}^s_\pi$, as $\mathtt{M}^s_\pi$ is a ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)*-partial evaluation* of $\mathtt{M}_\pi$. Indeed, in an arbitrary partial evaluation of $\mathtt{M}_\pi$, the definition of `Statement` might be only partially evaluated, and hence it could not be removed without spoiling the equivalence with the initial program.

In the next two sections we show how an analysis of the given program expression `E` (used during the partial evaluation phase) can support more dramatic optimisations of the meta-interpreter.

4.2.1 Meta-level elimination

Let us consider first the cases in which the given program expression `E` does not contain `Enc` or `Import` operations. It is easy to observe that in such cases, by definition of Γ, the set $\Gamma(\mathtt{E})$ is empty. Therefore a ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)-partial evaluation of $\mathtt{M}_\pi$ w.r.t. $\mathbf{A}' = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in (\Gamma(\mathtt{E}) \cup \{\mathtt{E}\})\}$ is simply a ($R_{\mathtt{CS}}$,$\mathtt{S}_1$,$\mathtt{S}_2$)-partial evaluation of $\mathtt{M}_\pi$ w.r.t. the set $\{\mathtt{Clause(E,x)}\}$. As a consequence, in these cases, the partial evaluation phase returns a meta-interpreter $\mathtt{M}^s_\pi$ of the form:

```
Demo(e,Empty).
Demo(e,(y And z)) <- Demo(e,y) & Demo(e,z).
Demo(e,y) <- Clause(e,y If z) & Demo(e,z).
Clause(E, H1 If B1).
...
Clause(E, Hn If Bn).
```

It is important to observe that in the above program the definition of `Clause` does not depend on the definition of `Demo` (as in the original program $\mathtt{M}_\pi$). Actually the above program corresponds to a standard vanilla meta-interpreter for a single object program, in which the first argument (viz., the program expression) of both `Demo` and `Clause` can be removed altogether. Such a correspondence suggests a major optimisation step, which is formally justified by the equivalence between the vanilla meta-interpreter and the interpreted object program, recently established by Martens and De Schreye [27]. This equivalence result can be stated as follows.

Proposition 5 *Let* $\mathtt{P}$ *be a definite object program, and let* $\mathtt{M}_\pi$ *be the vanilla meta-interpreter augmented with the (non-ground) meta-level representation of* $\mathtt{P}$ *(viz.,* $\pi = \{\mathtt{P}\}$*). Then, for any object level atom* A*, the following hold:*

(i) $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Demo(P,A)}\}$ *has a SLD-refutation with computed answer* ϑ *iff* $\mathtt{P} \cup \{\leftarrow \mathtt{A}\}$ *does, and*

(ii) $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Demo(P,A)}\}$ *has a finitely failed SLD-tree iff* $\mathtt{P} \cup \{\leftarrow \mathtt{A}\}$ *does.*

Proof.
(i) is simply a reformulation of theorem 7.3 in [27], where the equivalence is stated in model-theoretic terms, i.e. on the S-models [12] of the two programs. The proof-theoretic correspondent holds by virtue of the equivalence between S-models and computed answer substitutions [12].
(ii) descends, like (i), from the correspondence between object level derivations (in P) and meta-level derivations (in $\mathtt{M}_\pi$). Notice that arbitrary selections of atoms in meta-level derivations may cause infinite derivations for atoms for which every object level derivation is finite. Namely this may happen if clause (2) and clause (3) of $\mathtt{M}_\pi$ are applied an infinite number of times to atoms of the form `Demo(P,x)` with `x` uninstantiated. Therefore (ii) holds provided that such a situation does not occur, for instance for computation rules $\mathcal{R}_{\mathtt{Clause}}$ (giving priority to `Clause`). More generally, (ii) holds for any computation rule for $\mathtt{M}_\pi$ that selects only partially instantiated atoms. (For instance, such a constraint can be expressed in Gödel by means of delay declarations of the form

```
DELAY Demo(_, x) UNTIL NONVAR(x).
```
□

The equivalence between the vanilla meta-interpreter and the interpreted object program [8, 27] allows us to transform the specialised meta-interpreter $\mathtt{M}^s_\pi$ reported at the beginning of this section into an equivalent *object* program of the form:

```
H1 <- B1.
...
Hn <- Bn.
```

Therefore, in these cases the specialisation process eliminates not only the overhead due to the handling of program compositions but also the whole meta-interpretation layer overhead. As we shall discuss in Sect. 5, this corresponds to a speed-up factor of about 10 when moving from the extended meta-interpreter $\mathtt{M}_\pi$ to the equivalent object program.

Example 4 Referring back to example 1 introduced in Sect. 2, consider the extended vanilla meta-interpreter $\mathtt{M}_\pi$ (clauses (1)–(9)) with the clauses defining the set of object programs π = {`Train`, `Path`, `IC`}. The specialisation of this program $\mathtt{M}_\pi$ w.r.t. the expression `E` = `Pr(Path) Union (Pr(Train) Inters Pr(IC))` yields the following program, equivalent to $\mathtt{M}_\pi$.

```
Demo(e, Empty).
Demo(e, y And z) <- Demo(e,y) & Demo(e,z) .
Demo(e,y) <- Clause(e,y If z) & Demo(e,z).
Clause(Pr(Path) Union (Pr(Train) Inters Pr(IC)),
         Path(city1,city2) If Train(city1,city2,type).
Clause(Pr(Path) Union (Pr(Train) Inters Pr(IC)),
         Path(city1,city2) If Train(city1,city3,type) And Path(city3,city2).
Clause(Pr(Path) Union (Pr(Train) Inters Pr(IC)), Train(Florence,Rome,Int)).
Clause(Pr(Path) Union (Pr(Train) Inters Pr(IC)), Train(Milan,Florence,Int)).
```

The above meta-program can be then flattened down at the object level, obtaining the following equivalent object program:

```
Path(city1,city2) <-  Train(city1,city2,type).
Path(city1,city2) <-  Train(city1,city3,type) & Path(city3,city2).
Train(Florence,Rome,Int).
Train(Milan,Florence,Int).
```

□

4.2.2 Program expression argument elimination

Let us now consider the cases in which the given program expression `E` contains `Enc` or `Import` operations. It is easy to observe that, differently from the cases analysed in the previous subsection, now the set of program expressions $\Gamma(\texttt{E})$ is not empty. Therefore a ($R_{\texttt{CS}}$,$\texttt{S}_1$,$\texttt{S}_2$)-partial evaluation of $\texttt{M}_\pi$ w.r.t. **A'** = $\{\texttt{Clause(F,x)} \mid \texttt{F} \in (\Gamma(\texttt{E}) \cup \{\texttt{E}\})\}$ will contain some definition of `Clause` which does not consist of unit clauses only. For instance, if `F` belongs to $\Gamma(\texttt{E})$ then a ($R_{\texttt{CS}}$,$\texttt{S}_1$,$\texttt{S}_2$)-partial evaluation of $\texttt{M}_\pi$ w.r.t. **A'** may contain a statement of the form:

```
Clause(Enc(F), A If Empty) <- Demo(F, A).
```

In these cases, the specialised meta-interpreter $\texttt{M}^s_\pi$ cannot be directly transformed into an equivalent object program, as in the previous section. It is however possible to optimise $\texttt{M}^s_\pi$ by eliminating the program expression argument (viz., the first argument) from both predicates `Demo` and `Clause`, thus obtaining a new meta-program $\texttt{New\_M}_\pi$. The transformation $\texttt{PE\_Elim(M}^s_\pi\texttt{,E)} = \texttt{New\_M}_\pi$ is defined by the following steps:

(1) Let $\{\texttt{F0}, \texttt{F1}, \ldots, \texttt{Fm}\} = (\Gamma(\texttt{E}) \cup \{\texttt{E}\})$, where $\texttt{F0} = \texttt{E}$. For each program expression `Fi`, two new predicates `Demo_i` and `Clause_i` are introduced.

(2) Each atom of the form `Clause(Fi,t)` (resp., `Demo(Fi,t)`) in the definition of `Clause` is replaced by an atom of the form `Clause_i(t)` (resp., `Demo_i(t)`).

(3) The definition of predicate `Demo` is replaced by the definition of `m+1` predicates `Demo_0`, `Demo_1`, ..., `Demo_m`, each of them defined as follows:

```
Demo_i(Empty).
Demo_i(y And z) <- Demo_i(y) & Demo_i(z).
Demo_i(y) <- Clause_i(y If z) & Demo_i(z).
```

(4) Finally, the bridge clause:

```
Demo(E,y) <- Demo_0(y).
```

is introduced.

The above transformation defines a standard argument elimination technique that sensibly reduces the execution time needed to perform unifications [15, 32]. Indeed, even though the transformation slightly increases the size of the code, it provides a considerable performance improvement, as we will discuss in Sect. 5.

Example 5 Referring back to examples 1 and 4, the partial evaluation phase, applied to $\mathtt{M}_\pi$ w.r.t. the expression `E= Pr(Path) Union Enc(Pr(Train) Inters Pr(IC))`, yields the program $\mathtt{M}_\pi^s$:

```
Demo(e,Empty).
Demo(e,y And z) <-  Demo(e,y) & Demo(e,z).
Demo(e,y) <- Clause(e,y If z) & Demo(e,z).
Clause(Pr(Path) Union Enc(Pr(Train) Inters Pr(IC)),
                Path(x,y) If Train(x,w,t) And Path(w,y)).
Clause(Pr(Path) Union Enc(Pr(Train) Inters Pr(IC)),
                Path(x,y) If Train(x,y,t)).
Clause(Pr(Path) Union Enc(Pr(Train) Inters Pr(IC)), z If Empty) <-
                Demo(Pr(Train) Inters Pr(IC), z).
Clause(Pr(Train) Inters Pr(IC), Train(Florence,Rome,Int) If Empty ).
Clause(Pr(Train) Inters Pr(IC), Train(Milan,Florence,Int) If Empty ).
```

The application of the program transformation `PE_Elim(`$\mathtt{M}_\pi^s$`,E)` then produces the new program `New_`$\mathtt{M}_\pi$:

```
Demo(Pr(Path) Union Enc(Pr(Train) Inters Pr(IC)),y) <-  Demo_0(y).
Demo_0(Empty).
Demo_0(y And z) <-  Demo_0(y) & Demo_0(z).
Demo_0(y) <- Clause_0(y If z) & Demo_0(z).
Clause_0(Path(x,y) If Train(x,w,t) And Path(w,y)).
Clause_0(Path(x,y) If Train(x,y,t)).
Clause_0(z If Empty) <- Demo_1(z).
Demo_1(Empty).
Demo_1(y And z) <-  Demo_1(y) & Demo_1(z).
Demo_1(y) <-  Clause_1(y If z) & Demo_1(z).
Clause_1(Train(Florence,Rome,Int) If Empty ).
Clause_1(Train(Milan,Florence,Int) If Empty ).
```

□

The correctness of the transformation `PE_Elim` descends from the one-to-one correspondence between derivations in $\mathtt{M}_\pi^s$ and derivations in the transformed program.

Proposition 6 *Let* $\mathtt{E}$ *be a program expression, and let* $\mathbf{A}' = \{\mathtt{Clause(F,x)} \mid \mathtt{F} \in (\Gamma(\mathtt{E}) \cup \{\mathtt{E}\})\}$*. Let* $\mathtt{M}^s_\pi$ *be a* $(R_{\mathtt{CS}},\mathtt{S}_1,S_2)$ *partial evaluation of* $\mathtt{M}_\pi$ *w.r.t.* $\mathbf{A}'$*, and let* $\mathtt{New\_M}_\pi = \mathtt{PE\_Elim}(\mathtt{M}^s_\pi,\mathtt{E})$*. Then, for any object level query* $\mathtt{G}$*, the following hold:*

(i) $\mathtt{New\_M}_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *has a SLD-refutation with computed answer* ϑ *iff* $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *does,*

(ii) $\mathtt{New\_M}_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *has a finitely failed SLD-tree iff* $\mathtt{M}_\pi \cup \{\leftarrow \mathtt{Demo(E,G)}\}$ *does.*

Proof.
Proposition 4 (Sect. 4) establishes the equivalence of $\mathtt{M}^s_\pi$ and $\mathtt{M}_\pi$ w.r.t. queries of the form `Demo(E,G)`. The equivalence between $\mathtt{M}^s_\pi$ and $\mathtt{New\_M}_\pi$ descends immediately from the following observation, which can be easily proved by induction on the length of the derivation.
Let $\{\mathtt{F0},\mathtt{F1},\ldots,\mathtt{Fm}\} = (\Gamma(\mathtt{E}) \cup \{\mathtt{E}\})$, where $\mathtt{F0} = \mathtt{E}$. Then $d = \gamma_1,\gamma_2,\ldots,\gamma_n$ is a derivation for `Demo(E,G)` in $\mathtt{M}^s_\pi$ *if and only if* $d' = \gamma_1,\gamma'_1,\gamma'_2,\ldots,\gamma'_n$ is a derivation for `Demo_0(G)` in $\mathtt{New\_M}_\pi$, where each γ'_i is obtained from γ_i by replacing each atom `Clause(F,t)` by `Clause_i(t)`, and each `Demo(Fi,t)` by `Demo_i(t)`. □

4.3 Implementation

We now briefly discuss the actual implementation of the program specialiser described in the previous sections. The specialiser, given the meta-interpreter $\mathtt{M}_\pi$ and a program expression `E`, builds a specialised version $\mathtt{New\_M}_\pi$ of $\mathtt{M}_\pi$ by performing first a partial evaluation phase (described in Sect. 4.1) and then a program transformation phase (described in Sect. 4.2).

Phase I: The set of program expressions $\Gamma(\mathtt{E})$ is computed by means of a syntactic analysis of the given expression `E`. The $(R_{\mathtt{CS}},\mathtt{S}_1,\mathtt{S}_2)$-partial evaluation of `Clause(F,x)` in $\mathtt{M}_\pi$ is then built separately — for each $\mathtt{F} \in (\Gamma(\mathtt{E}) \cup \{\mathtt{E}\})$ — by applying a finite number of unfolding steps. The first step consists of collecting the set `S` of (suitably instantiated) clauses of $\mathtt{M}_\pi$ whose heads unify with the atom `Clause(F,x)`. The other steps consist of selecting a clause `C` from `S` and an atom `B` from the body of `C` with predicate `Clause` or `Statement`. The set `S` is then updated by substituting `C` with a (possibly empty) set of clauses, each obtained from `C` by replacing atom `B` with the (suitably instantiated) body of a clause from $\mathtt{M}_\pi$ whose head unifies with `B`. If no clause in `S` contains an atom `B` in its body with predicate `Clause` or `Statement`, the unfolding process terminates. All the obtained partial evaluations are then grouped together, along with the original clauses (1)—(3) defining `Demo` in $\mathtt{M}_\pi$, to obtain the partially evaluated meta-program $\mathtt{M}_\pi^s$.

Phase II: The program transformation phase applies one of the two transformations described in Sect. 4.2.1 and 4.2.2, depending on the structure of the given program expression `E`. If the set of expressions $\Gamma(\mathtt{E})$ is empty then a new object program is built by means of a transformation which is the inverse of the transformation used for building the meta-level representation of an object program [5]. Namely, each statement defining `Clause` in $\mathtt{M}_\pi^s$ is transformed into an object level statement by converting some of the function symbols of $\mathtt{M}_\pi^s$ into predicate symbols. For instance the statement

```
Clause(E, P(x) If Q(x) And R(F(x))).
```

is transformed into

```
P(x) <- Q(x) & R(F(x)).
```

If, instead, the set of expression $\Gamma(\mathtt{E})$ is not empty then the transformation described in Sect. 4.2.2 is applied to $\mathtt{M}_\pi^s$ in order to build the specialised program $\mathtt{New\_M}_\pi$.

The whole program specialiser has been written in Gödel [20]. The choice of using Gödel was motivated by two different considerations. First, one of our objectives was to build a program specialiser to be integrated with the existing system `COMPOSE` [5], which is written in Gödel. Second, Gödel offers considerable support for the development of meta-programming applications, such as partial evaluators, adopting a declarative programming style [3, 5, 17]. Indeed, Gödel contains several systems modules which offer operations for generating and handling the ground representation of programs. Gödel's abstract data type view of modules is particularly powerful in the case of meta-programming. As pointed out also in [5], system modules such as `Programs` and `Syntax` support a declarative style of meta-programming by hiding (most of) the complexity of the ground representation.

Finally, it is worth observing that both kinds of representations of object level programs — ground and non-ground — are employed in our specialiser. Indeed the meta-interpreter $\mathtt{M}_\pi$ defining meta-level compositions of programs employs a *non-ground* representation of object programs. On the other hand, the meta-program defining the program specialiser employs a *ground* representation of the object program (which is in turn the meta-interpreter $\mathtt{M}_\pi$).

5 Assessment

The specialiser described in the previous sections, given a meta-interpreter $\mathtt{M}_\pi$ and a program expression `E`, always returns a specialised program $\mathtt{New\_M}_\pi$ that is more efficient than $\mathtt{M}_\pi$. We will now compare $\mathtt{M}_\pi$ and $\mathtt{New\_M}_\pi$ as far as memory occupation and execution times are concerned.

5.1 Memory occupation

When considering memory occupation, a typical risk of applying standard partial evaluation techniques is represented by code explosion [2, 9, 24]. In our case, as we shall discuss below, the partial evaluation strategy employed ensures that the size of the specialised program $\mathtt{New\_M}_\pi$ is approximately the same as the size of the initial program $\mathtt{M}_\pi$.

Let us first make some observations on the relation between the number of clauses of $\mathtt{M}_\pi$ and $\mathtt{M}^s_\pi$, the latter being the program derived by the partial evaluation phase.

(1) The definition of predicate `Demo` in $\mathtt{M}^s_\pi$ is exactly the same as in $\mathtt{M}_\pi$.

(2) The definition of predicate `Statement` is always removed altogether from $\mathtt{M}^s_\pi$.

(3) The number of statements forming the definition of `Clause` in $\mathtt{M}^s_\pi$ depends on the structure of the given program expression `E`.

- If `E` does not contain `Inters` operations, then the definition of `Clause` in $\mathtt{M}^s_\pi$ contains $m + c$ statements, where m is the size of the definition of `Statement` in $\mathtt{M}_\pi$ and c is the number of `Enc` operations occurring in `E`.
- If instead `E` contains `Inters` operations, then the exact size of the definition of `Clause` in $\mathtt{M}^s_\pi$ depends on the structure of the object programs in `E`.

  For instance, if `E` = `Pr(P) Inters Pr(Q)`, where `P` and `Q` are object programs, then the size of the definition of `Clause` may vary from 0 to $m*n$, where m and n are the numbers of statements in `P` and `Q`, respectively. Indeed, by clause (6) in the definition of `Demo`, we have that `Clause(Pr(P) Inters Pr(Q), A If Body)` belongs to $\mathtt{M}^s_\pi$ only if there is a clause `A1 <- B1` in `P` and a clause `A2 <- B2` in `Q` such that their heads unify via ϑ in `A` and `Body` = `(B1 And B2)`ϑ. However, on the average, the size of the definition of `Clause` in $\mathtt{M}^s_\pi$ is less than $m + n$ in these cases, because of the selection operated by the unification mechanism.

Summing up, we have that the size of $\mathtt{M}_\pi^s$ is approximatively the same as the size of $\mathtt{M}_\pi$, that is, the partial evaluation phase does not produce a code explosion. After the partial evaluation phase, the program specialiser performs a program transformation phase to yield the final specialised program $\mathtt{New\_M}_\pi$. As described in Sect. 4.2, two alternative transformations are applied depending on the structure of the given program expression E.

(1) If E does not contain Enc or Import operations, then $\mathtt{M}_\pi^s$ is transformed into an equivalent object program $\mathtt{New\_M}_\pi$. In this case $\mathtt{New\_M}_\pi$ contains as many clauses as $\mathtt{M}_\pi^s$, minus the three clauses defining Demo in $\mathtt{M}_\pi^s$.

(2) If E instead contains Enc or Import operations, then $\mathtt{New\_M}_\pi$ contains $3m + 1$ clauses more than $\mathtt{M}_\pi^s$, where m is the number of program expressions in $\Gamma(\mathtt{E})$.

Therefore the program transformation phase does not increase significantly the size of $\mathtt{M}_\pi^s$ and hence we have that the whole program specialiser yields a specialised program $\mathtt{New\_M}_\pi$ that has approximatively the same size as the initially given program $\mathtt{M}_\pi$.

5.2 Performance results

We now discuss the performance speed-up supported by our program specialiser by analysing some experimental results.

The results of applying our program specialiser to some benchmark examples are reported in Figure 2. The first column (Query) contains a representation of an object level query G. The second column (Prog_Exp) specifies the program expression E — where $\cup$ stands for Union, $\cap$ for Inters, $*$ for Enc and $\triangleleft$ for Import — w.r.t. which the meta-interpreter $\mathtt{M}_\pi$ is specialised. The third column ($\mathtt{M}_\pi$) contains the execution times for the query Demo(E,G) in the initial meta-interpreter $\mathtt{M}_\pi$. The fourth column ($\mathtt{New\_M}_\pi$) contains the execution times, for the same query, of the meta-interpreter $\mathtt{New\_M}_\pi$ obtained by specialising $\mathtt{M}_\pi$ w.r.t. E. The fifth column (Rate) explicitly shows the rate between the third and the fourth column, that is, the performance speed-up yielded by the specialisation process. Finally, the last column ($\mathtt{SAGE(M}_\pi\mathtt{)}$) contains the execution times of the program obtained by specialising $\mathtt{M}_\pi$ by means of SAGE [17], a partial evaluator for Gödel programs.

The first benchmark example is Quicksort, consisting of three object programs: Quick defining the kernel of the standard quicksort algorithm, Split defining the process of partitioning a list into two sub-lists, and the well-known program Append for concatenating two lists. The second example is the Hanoi problem, represented by means of an object program Hanoi, defining the solution strategy and by the Append program. The third example is a version of example 1 introduced in Sect. 2 extended to determine acyclic paths consisting only of international train connections between cities. The object programs employed are Path, Train and IC (analogous to those introduced in Sect. 2), and a program Lib defining standard auxiliary predicates. Execution times have been calculated with (Bristol) Gödel 1.4 and are expressed in milliseconds.

Query	Prog_Exp	M_π	New_M_π	Rate	SAGE(M_π)
Quicksort(50)	Quick ∪ Append ∪ Split	106.8	9.1	11.73	117.4
	(Quick ∪ Append) ◁ Split	191.2	121.8	1.56	245.8
Hanoi(10)	Hanoi ∪ Append	291.4	23.4	12.45	265.3
	Hanoi ∪ (Append)*	186.7	90.9	2.05	176.9
Ac_Path(Rome, Oslo)	Path ∪ Lib ∪ (Train ∩ IC)	781.6	10.0	78.16	76.2
	Path ∪ (Lib)* ∪ (Train ∩ IC)	513.4	370.0	1.38	79.25

Figure 2: Some experimental results.

The results reported in Figure 2 highlight that when the given program expression E does not contain Enc or Import operations, then our program specialiser produces an object level program thus yielding a speed-up factor greater than 10. It is worth observing that this factor may grow further if E contains Inters operations, which can sensibly reduce the size of the specialised program. When instead E contains Enc or Import operations, the specialised meta-interpreter cannot be flattened down to the object level, and therefore the corresponding speed-up is considerably smaller.

The last column in Figure 5.2 shows that, for the first two examples, the execution times of the meta-interpreters obtained by applying the partial evaluator SAGE to M_π are close to the execution times of the initial meta-interpreters. There are two main reasons for this behaviour. First, though `SAGE` can be applied to any Gödel program, it is primarily targeted to optimise (meta-)programs using the ground representation of object programs. Moreover, `SAGE` performs a static analysis of the initial program in order to distinguish *safe* predicates, whose unfolding is always terminating and which will be specialised, from *unsafe* predicates which will be only partly specialised. In the case of the meta-interpreter $\mathtt{M}_\pi$ for the first two examples, predicates `Clause` and `Demo` are marked unsafe by `SAGE`, while only `Statement` is marked safe and unfolded. Therefore, the specialised program is similar to the initial one, and presents similar performances. The third example (`Ac_Path`) is instead a case in which `SAGE` supports a very good speed-up. Namely, the very last row indicates a speed-up greater than 6, while the speed-up given by our strategy is around 1.4. The reason of this result is that while our strategy inhibits the unfolding of `Demo`, `SAGE` recognises in this case the possibility of a safe unfolding and performs it. This example also highlights how the structure of the program expression `E` employed to construct the query affects both the execution times of $\mathtt{M}_\pi$ and the results of the partial evaluation process. Indeed, if the program `Lib` (mainly consisting of unit clauses) is not encapsulated then our specialisation technique yields a speed-up that is around seven times better than what `SAGE` does.

6 Concluding remarks

Levi and Sardu [24] made one of the first experiments of applying partial evaluation techniques to multiple logic program settings. They describe the realisation of a partial evaluator for a knowledge base management system, where separate Prolog programs can be combined by means of inheritance operators. Their partial evaluator is based on controlled unfolding and is targeted to handle object programs written in full Prolog, including built-ins and cuts. The experimental results reported in [24] show that the partial evaluator is able to yield very good speed-ups for the examples considered. On the other hand, the correctness of the partial evaluation strategy employed is not discussed at all.

Bugliesi, Lamma and Mello [10] also studied the problem of applying partial evaluation techniques to specialising compositions of object logic programs in the contextual logic programming setting [28]. Their approach neatly differs from ours in that all the definitions and results for partial evaluation in logic programming presented in [26] are extended in [10] to deal with the extended multi-program setting. Definitions of extended resultant, extended SLD-tree, and so on, are introduced, and the basic results given in [10] are proved in the new setting. Conversely, a meta-programming definition of composition operations over logic programs (like the one considered in this paper) has several advantages. First, the considered extension can be directly expressed from within logic programming itself, so that for instance a standard logic programming interpreter can be exploited to query meta-level composition of programs. Second, it is not necessary to extend the theory of partial evaluation in logic programming, since the whole extension is expressed in terms of standard logic (meta-)programs and the results of [26] can be directly exploited, as shown in Sect. 4.

The problems of partially evaluating meta-programs interpreting a single object program have been studied by many authors, e.g. see [14, 29, 35]. These works extensively deal with the use of Prolog's non-logical features for writing partial evaluators. Gallagher [14] highlights the advantages of using general program specialisation techniques rather than only partial evaluation. For instance, it is shown how predicate creation and deletion can produce substantial efficiency improvements. It is also shown how complex control strategies may be compiled, by specialising meta-interpreters, into object programs. This technique is shown to produce high speed-ups in many cases, such as for coroutining problems. Other works address more general issues, such as the comparison of different unfolding strategies [2], termination properties of unfolding [9] and the partial evaluation of concurrent logic programs [31].

The specialisation of (meta-)interpreters has a strong relation with standard compilation techniques, as it was originally illustrated by Futamura [13] (*first Futamura projection*) and then also for instance in [14, 18, 34, 35]. In our case, program specialisation relates the meta-logical implementation of program composition operations (described in Sect. 2) with the compilation-oriented implementation described [4]. For instance, when the given program expression `E` does not contain `Enc` or `Import` operations, the specialisation process yields an object program which coincides with the program obtained via compilation according to [4].

Our specialisation technique is able to specialise the meta-interpreter $\mathtt{M}_\pi$ w.r.t. queries of the form `Demo(E,x)`, where the object level query `x` is not specified. The resulting specialised program $\mathtt{New\_M}_\pi$ can be then employed to query any object level goal `G` on the program expression `E`. The results presented in Sect. 5 (Figure 2) confirm the effectiveness of our program specialiser, and relate it with the use of `SAGE`.

As pointed out in the introduction, general existing program specialisation techniques can be employed to further specialise the obtained program $\texttt{New\_M}_\pi$ w.r.t. partially instantiated object queries. In this perspective, it is interesting to consider the possibility of combining our program specialiser with other existing general partial evaluators. In particular, we have experimented the possibility of using Mixtus [32], a partial evaluator for full Prolog that is able to specialise any Prolog program, and SP [16], a more general system that is able to specialise declarative logic programs, written according to Prolog's syntax. In order to evaluate the usefulness of combining our program specialisation technique with Mixtus or SP, we have compared the specialised programs obtained, respectively:

(I) By applying our program specialiser to the meta-interpreter $\texttt{M}_\pi$ w.r.t. a program expression E, by converting the specialised (definite) program $\texttt{New\_M}_\pi$ from the Gödel syntax into the Prolog syntax, and finally by specialising it by means of Mixtus (or SP) w.r.t. some partially instantiated object query G, or

(II) By employing only Mixtus (or SP) to specialise $\texttt{M}_\pi$ first w.r.t. to E, and then w.r.t. G.

Some experiments on the examples reported in Figure 2 have shown that the use of our program specialiser together with Mixtus (or SP) — approach (I) — gives better speed-ups than using Mixtus (or SP) alone — approach (II). More precisely, when our program specialiser succeeds in transforming the meta-interpreter $\texttt{M}_\pi$ into an equivalent object level program, the obtained speed up is definitively higher than the overall speed-up supported by approach (II). When, instead, our program specialiser cannot produce an object level program, then the two approaches yield very similar results.

It is worth noting that, if we consider only the first stage of the specialisation process (i.e., the specialisation w.r.t. E) then Mixtus and SP may give better results than our specialiser. This is due to the fact that Mixtus and SP employ more sophisticated program specialisation techniques than our simple (and partial) unfolding. However, after Mixtus or SP have been applied to perform the second stage of the specialisation process (w.r.t. G), the combined approach (I) offers a better speed-up than using only Mixtus or SP (II). A disadvantage of combining our specialiser with Mixtus or SP concerns declarativeness. Namely, the declarative style of our program specialiser — using Gödel ground representation of object programs — is not supported by Mixtus and SP, which employ Prolog's non-declarative features.

It is important to observe that our program specialisation technique can be exploited to deal with other significant extensions of the multi program setting described in Sect. 2. Namely, the specialisation technique described in this paper has been also applied to meta-level compositions of object programs written in an extended syntax. As shown in [7], allowing object level programs to contain formulae of the form `G In E`, where `G` is an object level goal and `E` is a program expression, notably increases the expressive power of the language of program expressions, for instance by supporting forms of context-switching and message-passing. The meaning of the meta-annotation `In` is defined by adding the following clause to the extended vanilla meta-interpreter:

```
(3') Demo(e, (g In e1)) <- Demo(e1, g).
```

The only change needed to apply our specialisation technique to this extension concerns the set of program expressions w.r.t. which $\mathtt{M}_\pi$ must be partially evaluated. Namely, given a program expression `E`, the set of "critical" program expressions includes — in addition to the set $\Gamma(\mathtt{E}) \cup \{\mathtt{E}\}$ — all program expressions occurring in the object programs in `E`. Formally, this corresponds to refining the definition of Γ in the case of plain programs as follows:

$\Gamma(\mathtt{P}) = \{\mathtt{F} \mid$ a goal (`G In F`) occurs in the object program $\mathtt{P}\}$.

Summing up, we have presented a specialisation technique for handling meta-interpreters defining compositions of object programs. The correctness of the transformation has been proved by resorting to standard results of partial evaluation in logic programming. The program specialiser has been implemented in Gödel by using meta-programming and a ground representation of the object program (i.e., the extended vanilla meta-interpreter $\mathtt{M}_\pi$), so that the whole program specialiser is written in a declarative way. The effectiveness of the approach has been demonstrated by comparing the speed-up offered by our program specialiser with the speed-up obtained by applying `SAGE`, as well as other existing specialisers for Prolog.

An interesting direction for future work is the use of different representations of object programs for implementing program specialisers. Leuschel and Martens [23], for instance, have recently pointed out the disadvantages of writing partial evaluators by employing a ground representation of object programs in which (the representation of) substitutions must be explicitly handled. They have therefore proposed a new ground representation for object (Prolog) programs in which the representation of substitutions is hidden at the meta-level. The results reported in [23] indicate such a representation as very promising, at least for some applications.

Another interesting direction for future work is to introduce reflection mechanisms, as for instance those studied in [1, 11, 22], in order to integrate meta-level and object level derivations in the specialised programs. This could allow us to further improve the speed-up even when the specialised program cannot be directly converted into an equivalent object level program.

Acknowledgements

This work has been partly supported by the INTAS project 93-1702 on "Efficient Symbolic Computing".

References

[1] J. Barklund. Metaprogramming in Logic. Technical report, Uppsala University, 1994.

[2] K. Benkerimi and J.W. Lloyd. A Partial Evaluation Procedure for Logic Programs. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990*, pages 343–358. MIT, 1990.

[3] A. F. Bowers and C. A. Gurr. Towards Fast and Declarative Meta-Programming. In K.R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, chapter 6, pages 137–166. The MIT Press, 1995.

[4] A. Brogi. *Program Construction in Computational Logic*. PhD thesis, University of Pisa, 1993.

[5] A. Brogi and S. Contiero. Composing logic programs by meta-programming in Gödel. In K. Apt and F.Turini, editors, *Meta-programming and Logic Programming*, pages 167–194. The MIT Press, 1995.

[6] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, 1994.

[7] A. Brogi, C. Renso, and F. Turini. Amalgamating Language and Meta-Language for Composing Logic Programs. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proceedings of the GULP-PRODE'94 Joint Conference on Declarative Programming*, volume 2, pages 408–422, 1994.

[8] A. Brogi and F. Turini. Meta-logic for program composition: Semantic issues. In K.R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, chapter 4, pages 83–110. The MIT Press, 1995.

[9] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Evaluation. *New Generation Computing*, 11(1):47–79, 1992.

[10] M. Bugliesi, E. Lamma, and P. Mello. Partial Deduction for Structured Logic Programming. *Journal of Logic Programming*, 16(1, 2):89–122, 1993.

[11] S. Costantini, P. Dell'Acqua, and G. A. Lanzarone. Extending Horn Clauses Theories by Reflection Principles. In M. C. MacNish, D. Pearce, and L. M. Pereira, editors, *Lecture Notes in Artificial Intelligence n.838*, pages 400–413. Springer-Verlag, 1994.

[12] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[13] Y. Futamura. Partial Computation of Programs. In J. Hartmanis G. Goos, editor, *RIMS Simposia on Software Science and Engineering*, number 147 in LNCS, pages 1–35. Springer-Verlag, 1982.

[14] J. Gallagher. Transforming Logic Programs By Specialising Interpreters. In *ECAI-86, Proceedings of the 7th European Conference on Artificial Intelligence, Brighton, England*, pages 313–326, 1986.

[15] J. Gallagher and M. Bruynooghe. Some Low-Level Source Transformations for Logic Programs. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic, Katholieke Universiteit Leuven, Belgium*, pages 229–244, 1990.

[16] J.P. Gallagher. A System For Specialising Logic Programs. Technical report, University of Bristol, 1991.

[17] C.A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, University of Bristol, 1994.

[18] P.M. Hill and J. Gallagher. Meta-Programming in Logic Programming. In *Artificial Intelligence and Logic Programming.* Oxford University Press, 1994.

[19] P.M. Hill and J.W. Lloyd. Analysis of metaprograms. In H.D. Abramson and M.H. Rogers, editors, *Metaprogramming in Logic Programming*, pages 23–52. The MIT Press, 1989.

[20] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* The MIT Press, 1994.

[21] J. Komorowski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta-Programming in Logic, Uppsala, Sweden, June 1992 (Lecture Notes in Computer Science, vol. 649)*, pages 49–69. Springer Verlag, 1992.

[22] E. Lamma, P. Mello, and A. Natali. Reflection Mechanisms for Combining Prolog Databases. *Software: Practice and Experience*, 21(6):603–624, 1991.

[23] M. Leuschel and B. Martens. Partial Deduction of the Ground Representation and its Application to Integrity Checking. In John Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 495–509. MIT Press, 1995.

[24] G. Levi and G. Sardu. Partial Evaluation of Metaprograms in a "Multiple Worlds" Logic Language. *New Generation Computing*, 6(2,3):227–247, 1988.

[25] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, second edition, 1987.

[26] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[27] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *Journal of Logic Programming*, 22(1):47–99, 1995.

[28] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proceedings Sixth International Conference on Logic Programming*, pages 284–302. The MIT Press, 1989.

[29] S. Owen. Issues in the Partial Evaluation of Meta-Interpreters. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 319–340. MIT, 1989.

[30] A. Pettorossi and M. Proietti. Transformations of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19 & 20:261–320, 1994.

[31] S. Safra and E. Y. Shapiro. Meta interpreters for real. In H.-J. Kugler, editor, *Information Processsing 86*, pages 271–278. North-Holland, 1986.

[32] D. Shalin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, The Royal Institute of Technology (KTH), Department of Telecommunications and Computer Systems, Stockolm, Sweden, 1991.

[33] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[34] A. Takeuchi. Affinity between meta interpreters and partial evaluation. In H.-J. Kugler, editor, *Information Processing 86*, pages 279–282. North-Holland, 1986.

[35] A. Takeuchi and K. Furukawa. Partial evaluation of prolog programs and its application to meta-programming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420. North-Holland, 1986.