

# To Parse or Not To Parse

*Wim Vanhoof*      *Bern Martens*

*Report CW 251, June 1997*

Department of Computing Science, K.U.Leuven

## **Abstract**

Writing meta interpreters is a well-known technique to enhance the expressive power of logic programs. However, the resulting interpretation overhead considerably slows down program execution. A natural approach to solving this efficiency problem consists in specialising the interpreter with respect to a given object program, thus removing the overhead. Fully achieving the latter goal however, turns out to be a non trivial task. Satisfying results could often only be obtained at the cost of using not fully automatic and/or ad hoc techniques. In this paper, we reconsider the problem of specialising the vanilla meta interpreter through *fully automatic and completely general* methods. In particular, we study how the homeomorphic embedding relation guides specialisation of the interpreter. We focus on the so-called *parsing problem*, i.e. in essence removing meta-interpretation overhead, in particular parsing conjunctive goals, and demonstrate that further control refinements are necessary to properly deal with it. In particular, we modify *local* control on the basis of information imported from the *global* level. Subsequently, we use this technique to specialise an extended meta interpreter, dealing with compositions of logic programs, and obtain satisfying results.

# To Parse or Not To Parse

Wim Vanhoof \*      Bern Martens †

Departement Computerwetenschappen  
Katholieke Universiteit Leuven  
Celestijnenlaan 200A, B-3001 Heverlee, Belgium  
e-mail : {wimvh,bern}@cs.kuleuven.ac.be

## Abstract

Writing meta interpreters is a well-known technique to enhance the expressive power of logic programs. However, the resulting interpretation overhead considerably slows down program execution. A natural approach to solving this efficiency problem consists in specialising the interpreter with respect to a given object program, thus removing the overhead. Fully achieving the latter goal however, turns out to be a non trivial task. Satisfying results could often only be obtained at the cost of using not fully automatic and/or ad hoc techniques. In this paper, we reconsider the problem of specialising the vanilla meta interpreter through *fully automatic and completely general* methods. In particular, we study how the homeomorphic embedding relation guides specialisation of the interpreter. We focus on the so-called *parsing problem*, i.e. in essence removing meta-interpretation overhead, in particular parsing conjunctive goals, and demonstrate that further control refinements are necessary to properly deal with it. In particular, we modify *local* control on the basis of information imported from the *global* level. Subsequently, we use this technique to specialise an extended meta interpreter, dealing with compositions of logic programs, and obtain satisfying results.

## 1 Introduction

Writing meta interpreters is a well-known technique to enhance the expressive power of logic programs (see e.g. [36, 1, 18]). However, the resulting interpretation overhead considerably slows down program execution [4].

A natural approach to solving this efficiency problem consists in specialising the interpreter with respect to a given object program [12, 37, 32], thus removing the overhead. Fully achieving the latter goal however, turns out to be a non trivial task [4, 28]. Satisfying results could often only be obtained at the cost of using not fully automatic and/or ad hoc techniques [20]. In a recent paper [7], Brogi and Contiero address specialisation of a non-ground *vanilla*-like meta interpreter [35, 19, 36, 29], extended in order to deal with program compositions [8, 9, 5, 7]. Again, the interpretation overhead is removed by an ad hoc specialiser, specifically constructed to handle the extended interpreter.

On the other hand, research on partial deduction of logic programs meanwhile resulted in *fully automatic and completely general* specialisation techniques, not requiring any specific a priori knowledge about the kind of programs to be specialised, nor any help from the programmer (e.g. [14, 11, 24]).

---

\*Supported by a specialisation grant of the Flemish Institute for the Promotion of Scientific-Technological Research in Industry (IWT), Belgium.

†Postdoctoral Fellow of the K.U.Leuven Research Council, Belgium.

In this paper, we reconsider the problem of specialising the vanilla meta interpreter through such fully automatic methods. In particular, we study how the homeomorphic embedding relation guides specialisation of the interpreter. We focus on the so-called *parsing problem*, i.e. in essence removing meta-interpretation overhead, in particular parsing conjunctive goals [20, 28], and demonstrate that further control refinements are necessary to properly deal with it. In particular, we modify *local* control on the basis of information imported from the *global* level. The resulting control strategy, while remaining fully general, leads to excellent specialisation of vanilla like meta programs. Parsing is always specialised, but - appropriately, as we will show - not always completely removed. Finally, as a concrete application, we subject the extended vanilla meta interpreter of [7] to our techniques, showing we get at least the same degree of specialisation as in [7], and often even more.

The outline of the paper is as follows. In Section 2, we briefly recapitulate some aspects of the automatic partial deduction strategy we use, largely referring to the literature for further and more general information. Section 3 considers specialisation of the vanilla meta interpreter and constitutes the main body of this paper. Subsequently, in Section 4, specialisation of the program composition interpreter in [7] is addressed. We conclude with a discussion of our results and the ensuing plans for further research in Section 5. Throughout this paper, we only consider definite logic programs.

## 2 Automatic Partial Deduction

We assume the reader is familiar with the basic correctness [27] and control [14, 11, 25] notions of automatic (on-line) partial deduction (see also Appendix A). In this section, we briefly describe the essential ingredients of the partial deduction method we use in this paper. Experiments were conducted using the ECCE automatic partial deduction system [26].

### 2.1 Local control

As a starting point for local control, we use the one proposed in Section 4.2 of [17], partially imported by [34, 24]. The following definitions are taken from that paper.

**Definition 2.1** Given an SLD-tree  $\tau$ . Let  $G = \leftarrow A_1 \wedge \dots \wedge A_n$  be a goal in  $\tau$ ,  $A_m$  the selected atom in  $G$ ,  $A \leftarrow A'_1 \wedge \dots \wedge A'_k$  a clause of  $P$  such that  $\theta$  is an mgu of  $A_m$  and  $A$ . Then in  $\leftarrow (A_1 \wedge \dots \wedge A'_1 \wedge \dots \wedge A'_k \wedge \dots \wedge A_n)\theta$ , for each  $i \in \{1, \dots, k\}$ ,  $A'_i\theta$  descends from  $A_m$ , and for each  $i \in \{1, \dots, n\} \setminus \{m\}$ ,  $A_i\theta$  descends from  $A_i$ . If  $A'$  descends from  $A$ , and  $A''$  descends from  $A'$ , then  $A''$  also descends from  $A$ , i.e. the relation is transitive.

We define the *homeomorphic embedding relation*  $\sqsubseteq$  as follows. As usual,  $e_1 < e_2$  denotes that  $e_2$  is a strict instance of  $e_1$ .

**Definition 2.2** Let  $X, Y$  range over variables,  $f$  over functors, and  $p$  over predicates. Define  $\sqsubseteq$  on terms and atoms:

$$\begin{aligned} X &\sqsubseteq Y \\ s &\sqsubseteq f(t_1, \dots, t_n) && \Leftarrow s \sqsubseteq t_i \text{ for some } i \\ f(s_1, \dots, s_n) &\sqsubseteq f(t_1, \dots, t_n) && \Leftarrow s_i \sqsubseteq t_i \text{ for all } i \\ p(s_1, \dots, s_n) &\sqsubseteq p(t_1, \dots, t_n) && \Leftarrow s_i \sqsubseteq t_i \text{ for all } i \text{ and } p(t_1, \dots, t_n) \not\prec p(s_1, \dots, s_n) \end{aligned}$$

For atoms  $A, B$ :  $A \triangleleft B$  iff  $A \sqsubseteq B$  and  $A \not\approx B$  (where  $A \approx B$  denotes that  $A$  and  $B$  are variants).

**Definition 2.3** An atom  $A$  in a goal at the leaf of an SLD-tree is *selectable* unless it descends from a selected atom  $A'$ , with  $A' \sqsubseteq A$ .

**Definition 2.4** The *unfolding rule*  $U_{\triangleleft}$  unfolds the left-most selectable atom in each goal  $G$  of the SLD-tree under construction. If no atom is selectable, no further unfolding is performed.

Independently of whether it is selectable according to definition 2.3, often we will denote the selected atom (under any unfolding rule) in a goal  $G$  by  $s(G)$ , leaving the unfolding rule implicit.

An important observation is that  $U_{\triangleleft}$  always terminates [17]. For a program  $P$  and a goal  $G$ ,  $\tau_U(P, G)$  will denote the SLD-tree built by unfolding rule  $U$  for  $G$  in  $P$ , often simply written as  $\tau_U(G)$  or  $\tau_G$ . Given an SLD-tree  $\tau$ ,  $AL(\tau)$  denotes the set of atoms in its leaves.

## 2.2 Global control

We adopt the framework for global control of [24, 25] and introduce some notation. For any program  $P$  and a goal  $G$ , partial deduction of  $P$  w.r.t.  $G$  leads to a global tree, denoted by  $\mathcal{G}(P, G)$ , containing the roots of the SLD-trees built locally. To ensure the coveredness condition of [27] in a maximally precise way, for atoms appearing in the leaf of an SLD-tree, a fresh such tree needs to be built when there is not yet one with (a variant of) the given atom as its root. However, in this basic form, this process often does not terminate. Hence, an atom  $A$  in the leaf of a freshly constructed SLD-tree  $\tau$  is added as a child of  $\tau$ 's root in the global tree, if there is no  $A' \in \mathcal{G}(P, G)$  (constructed so far) such that  $A' \approx A$ . Indeed, for such an atom (or a variant of it) no SLD-tree has yet been built. If  $A$  has an ancestor  $A' \in \mathcal{G}(P, G)$  such that  $charatom(A') \trianglelefteq charatom(A)$ , then  $A'$  is replaced by  $msg(A', A)$ , where  $charatom(A)$  denotes  $A$ 's characteristic atom (this generalisation ensures termination). For further details, we refer to [24, 25] and Appendix A. Below, we will often refer to a node  $G$  in a global tree, rather than immediately to the atom in such a node.

## 2.3 Code generation

Consider a program  $P$ , a goal  $G$  and the global tree  $\mathcal{G}(P, G)$  as a result of the partial deduction of  $P$  w.r.t.  $G$ . Then consider a node  $G_0 \in \mathcal{G}(P, G)$  with the associated SLD-tree  $\tau_{G_0}$ . For each leaf  $L$  of  $\tau_{G_0}$ , consisting of the atoms  $A_1, \dots, A_n$  a resultant clause is generated having the following form:  $G_0\theta \leftarrow A_1, \dots, A_n$  where  $\theta$  is the substitution built along the branch  $\leftarrow G_0 \dots \leftarrow A_1, \dots, A_n$ .

With each node  $G_0 \in \mathcal{G}(P, G)$  a renaming  $p_{G_0}(\overline{X})$  is associated, where  $p_{G_0}$  is a unique predicate name, and  $\overline{X}$  is the vector of free variables of  $G_0$ . Then each resultant clause  $C$ , defined as  $G_0\theta \leftarrow A_1, \dots, A_n$  is renamed to  $p_{G_0}\theta \leftarrow p_{G_1}\phi_1, \dots, p_{G_n}\phi_n$  where  $\forall i \in \{1 \dots n\}$ :  $A_i \approx G_i\phi_i$  for a substitution  $\phi_i$  such that  $\exists G' \in \mathcal{G}(P, G)$ :  $A_i \approx G'\phi'$  for a substitution  $\phi'$  and  $G_i \prec G'$ .

With  $PD(P, G)$  we denote the collection of renamed resultant clauses constituting the final partial deduction of  $P$  w.r.t.  $G$  using the techniques as introduced above.

## 3 Behaviour of $U_{\triangleleft}$ When Unfolding Meta Interpreters

In this section, we investigate automatic partial deduction of non-ground, vanilla meta programs, using the general technique presented in Section 2. We will in particular concentrate on the so called ‘‘parsing problem’’ [20, 28]: To what extent is our general technique able to remove meta-interpretation overhead, including the parsing of conjunctions in clause (2) of  $V$ .

Let us first repeat the definition of the well-known vanilla meta interpreter [35, 19, 36, 29].

**Definition 3.1** The *vanilla meta interpreter*  $V$ :

- (1)  $solve(true) \leftarrow$
- (2)  $solve((A, B)) \leftarrow solve(A), solve(B)$ .
- (3)  $solve(H) \leftarrow pclause(H, B), solve(B)$ .

As usual, an object program will be represented as a database of *pclause*-facts. Omitting the formal definition, we just include an example.

**Example 3.2** Consider an object program  $P$ , *reverse* with an accumulating parameter, where a type check on the accumulator is added [24], and its meta representation  $C(P)$ .

$$\begin{array}{ll}
reverse(L, R) \leftarrow rev(L, [], R) & pclause(reverse(L, R), rev(L, [], R)) \leftarrow \\
rev([], L, L) \leftarrow & pclause(rev([], L, L), true) \leftarrow \\
rev([X|X_s], L, R) \leftarrow ls(L), rev(X_s, [X|L], R). & pclause(rev([X|X_s], L, R), (ls(L), rev(X_s, [X|L], R))) \leftarrow \\
ls([]) \leftarrow & pclause(ls([], true) \leftarrow \\
ls([X|X_s]) \leftarrow ls(X_s). & pclause(ls([X|X_s]), ls(X_s)) \leftarrow
\end{array}$$

The combination of the interpreter,  $V$ , and the “encoded” object program,  $C(P)$ , will be denoted by  $V_P$ .

### 3.1 Removing Interpretation Overhead

For an object program  $P$ , we call *meta structure* all program structure in  $V_P$  that can never appear in arguments in  $P$ .<sup>1</sup> In concrete terms, this means the constant *true*, the functor  $/2$ , and all functors representing object level predicates.

**Definition 3.3** For any program  $P$  and goal  $G$ , the residual program,  $PD(V_P, G)$  is called *meta structure free* if for each renamed resultant clause  $p_{G_0}\theta \leftarrow p_{G_1}\phi_1, \dots, p_{G_n}\phi_n \in PD(V_P, G)$ ,  $\theta, \phi_1, \dots, \phi_n$  contains no meta structure.

We will show that by extending our unfolding rule, it can be proven that for any program  $P$  and goal  $G = solve(p(\overline{X}))$  where  $p$  denotes a predicate in  $P$ ,  $PD(V_P, G)$  is meta structure free<sup>2</sup>.

The following results are fairly obvious:

**Proposition 3.4** For any  $V_P$ , atoms with *pclause* as predicate are always unfolded by  $U_{\triangleleft}$ .

**Corollary 3.5** For any  $P$  and  $G$ ,  $\mathcal{G}(V_P, G)$  contains only *solve* atoms.

Obviously, in case no generalisations occur while constructing  $\mathcal{G}(V_P, G)$ , we have that for all  $G_0 \in \mathcal{G}(V_P, G)$  and for all  $A \in AL(\tau_{G_0})$  there exists a  $G' \in \mathcal{G}(V_P, G)$  such that  $A \approx G'$ . Since the same holds for the atom(s) in  $G$ , and *solve*( $X$ ) will not appear in any SLD-tree, code generation will produce a meta structure free residual program. We illustrate this with the partial deduction of Example 3.2.

**Example 3.6**  $P$  and  $C(P)$  are taken as in Example 3.2.  $PD(V_P, solve(reverse(L, R)))$  then looks as follows:

$$\begin{array}{l}
d_1([], []) \leftarrow \\
d_1([X_1|X_2], X_3) \leftarrow d_2([], X_2, X_1, X_3). \\
d_2([], [], X_1, [X_1]) \leftarrow \\
d_2([], [X_1|X_2], X_3, X_4) \leftarrow d_2([X_3], X_2, X_1, X_4). \\
d_2([X_1|X_2], [], X_3, [X_3, X_1|X_2]) \leftarrow d_3(X_2). \\
d_2([X_1|X_2], [X_3|X_4], X_5, X_6) \leftarrow d_3(X_2), d_2([X_5, X_1|X_2], X_4, X_3, X_6). \\
d_3([]) \leftarrow \\
d_3([X_1|X_2]) \leftarrow d_3(X_2).
\end{array}$$

<sup>1</sup> For more formal details, see [29].

<sup>2</sup> If  $G = solve(X)$ , the definition of the predicate, created from this root, will contain meta structure, in order to parse the top level goal. Although we do not consider this top level parsing to be a problem, we restrict the form of  $G$  in order to simplify the formulation of the theory and shorten the examples. Note however, that this restriction does not diminish the generality of our technique.

### 3.2 In Case Generalisations Take Place

$U_{\triangleleft}$  (and similar unfolding rules [30]) performs well when predicate arguments either shrink or grow throughout the unfolding process. Unfolding is allowed as long as information is consumed and appropriately halted when such is no longer the case.

Problems however arise for predicates handling *fluctuating* structure(s): structures that can grow, but also shrink between successive recursive calls. Most natural logic programs do not contain such predicates, but meta interpreters do. This is one of the main reasons why it is notoriously difficult to handle them well in automatic, general partial deduction.

Consider a clause of the following form:

$$p([X|X_s], A) \leftarrow a(X), p(X_s, [X|A])$$

If somewhere during the process,  $solve(p(X, A))$  is unfolded into  $solve(a(X'), p(X_s, [X'|A]))$ ,  $\triangleleft$ -embedding will be detected. If there is an atom  $solve(p(X, A))$  in the global tree, this atom will be generalised with  $solve(a(X'), p(X_s, [X'|A]))$ , resulting in  $solve(X)$ . The generalisation is justified, since on the object level, indeed  $p(X, A) \triangleleft p(X_s, [X'|A])$ . The problem is that the  $p$ -functor and all structure surrounding the embedded term are lost in the generalisation.

Now if  $solve(a(X'), p(X_s, [X'|A]))$  was unfolded by  $U_{\triangleleft}$  one step further, the atoms  $solve(a(X'))$  and  $solve(p(X_s, [X'|A]))$  would be brought onto the global level, and the generalisation would occur between  $solve(p(X, A))$  and  $solve(p(X_s, [X'|A]))$ , leading to  $solve(p(Y, Z))$  and leaving  $solve(a(X'))$  as it is.

In [20], it was noticed that *always* unfolding parsing calls seems a good idea, although no indication was given how to incorporate this idea in a general, automatic partial deduction technique. In subsection 3.3, we argue that, if we aim at obtaining a high degree of specialisation, it is often a better idea *not* to unfold parsing calls, at the cost of obtaining what we will call *specialised* parsing. We will therefore aim at further unfolding parsing calls if they lead to generalisation. To that end, we *modify our unfolding rule such that it takes global information into account*.

**Definition 3.7** The unfolding rule  $U_{ext}$  is applied to an atomic goal  $G_0$  as follows: If through (left-to-right) deterministic<sup>3</sup> unfolding of  $G_0$ , a derivation  $\leftarrow G_0 \dots \leftarrow G_n$  can be constructed such that  $\nexists i, j \in \{0 \dots n\}, i < j: s(G_j)$  descends from  $s(G_i)$ ,  $s(G_i) \triangleleft s(G_j)$  and  $G_n = \leftarrow A_1, \dots, A_m$  where  $\forall i \in \{1 \dots m\}: A_i \triangleleft G_0$ , then  $U_{ext}(G_0) = \leftarrow G_0 \dots \leftarrow G_n$ , where  $G_n$  is the first such goal found, else  $U_{ext}(G_0) = G_0$ .

Before introducing our extended unfolding rule,  $U_{\triangleleft}^+$ , we give some extra notation:  $V_{ext}(G_0)$  denotes the set of variables of  $G_0$ , denoted by  $Vars(G_0)$ , that are instantiated by  $U_{ext}$  in case  $U_{ext}(G_0) \neq G_0$ .

#### Algorithm 3.8

**Input:** a program  $P$ , goal  $G$ , a global tree  $\mathcal{G}(P, G)$  with  $A$  a leaf of  $\mathcal{G}(P, G)$ .

**Output:** SLD-tree  $\tau_{U_{\triangleleft}^+}(A)$ .

**Init:**  $i \leftarrow 1, \tau_0 \leftarrow \emptyset, \tau_1 \leftarrow \tau_{U_{\triangleleft}}(A)$ , Let  $Anc(A)$  be the ancestors of  $A$  ( $A$  included) in  $\mathcal{G}(P, G)$ .

**while**  $\tau_{i+1} \neq \tau_i$  **do**

$i \leftarrow i + 1$

**If**  $\exists$  a leaf  $L = \leftarrow A_1, \dots, A_n$  of  $\tau_i$  such that  $\exists A_j \in L, \exists B \in Anc(A)$  where

$B \triangleleft A_j, U_{ext}(A_j) \neq A_j$  and

$V_{ext}(A_j) \cap (Vars(A_1) \cup \dots \cup Vars(A_{j-1}) \cup Vars(A_{j+1}) \dots \cup Vars(A_n)) = \emptyset$ .<sup>4</sup> (\*)

**then**  $\tau_{i+1} \leftarrow \tau_i \uplus_{L,j} U_{ext}(A_j)$

**else**  $\tau_{i+1} \leftarrow \tau_i$

---

<sup>3</sup>using a look-ahead

<sup>4</sup>This condition is necessary for the present proof of Theorem 3.9 (see Appendix B). We are currently investigating to what extent it can be relaxed.

**end while**

$\tau_{U_{\sqsubseteq}^+}(A) \leftarrow \tau_i$

where  $\tau(A) \uplus_{L,j} \leftarrow G_0 \dots \leftarrow G_n$  is defined as extending the derivation

$\leftarrow A \dots \leftarrow L = \leftarrow A_1, \dots, A_{j-1}, G_0, A_{j+1}, \dots, A_n$  in  $\tau(A)$  into  $\leftarrow A \dots \leftarrow L = \leftarrow A_1, \dots, G_0, \dots, A_n \dots$   
 $\leftarrow A_1, \dots, A_{j-1}, G_n, A_{j+1}, \dots, A_n$ .

**Theorem 3.9** Algorithm 3.8 terminates and  $\tau_{U_{\sqsubseteq}^+}(A)$  is an SLD-tree for  $A$ .

The aim of enhancing  $U_{\sqsubseteq}$  into  $U_{\sqsubseteq}^+$  is clear: Avoiding generalisation at the global level when this can be safely achieved through some extra unfolding at the local level. Now, we can formulate an important result:

**Theorem 3.10** For any program  $P$ , goal  $G = solve(p(\overline{X}))$  with  $p$  a predicate in  $P$ , using the global control techniques described in Section 2 and the enhanced unfolding rule  $U_{\sqsubseteq}^+$ ,  $PD(V_P, G)$  is meta structure free.

### 3.3 Specialised Parsing

Opposite to observations in [20], where it is argued that (parsing) calls should always be unfolded,  $U_{\sqsubseteq}$  as well as  $U_{\sqsubseteq}^+$  will often stop unfolding at such calls:  $solve(C)$  where  $C$  is a meta representation of an object level conjunction. During code generation, a new predicate will be made from  $\tau_{solve(C)}$ , carrying all the (object) information of the conjunction. Consider the following example:

**Example 3.11**

$P : p(\square) \leftarrow$   
 $p([X|X_s]) \leftarrow q(X), p(X_s). \quad q(X) \leftarrow q(X).$

The most interesting SLD-tree branches, generated during derivation of  $PD(V_P, solve(p(X)))$  are depicted in Figure 1.

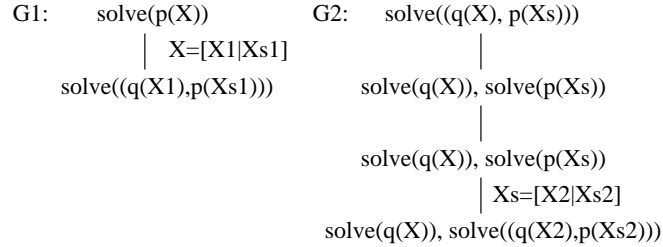


Figure 1: Branches of the first two SLD-trees constructed during generation of  $PD(V_P, solve(p(X)))$

If code is generated, by renaming  $G1$  to  $d_1$ ,  $G2$  to  $d_2$  and  $d_3$  is the specialised predicate for  $solve(q(X))$ , we get the following program:

$d_1(\square) \leftarrow \quad d_2(X, \square) \leftarrow d_3(X).$   
 $d_1([X|X_s]) \leftarrow d_2(X, X_s). \quad d_2(X, [Y|Y_s]) \leftarrow d_3(X), d_2(Y, Y_s).$   
 $d_3(X) \leftarrow d_3(X).$

Note how  $d_2$  carries out “specialised”, i.e. meta structure-less, parsing. If we unfold  $G1$  one step further, using  $V$ ’s parsing clause (2), we get the following program (renaming stays the same, although we have no predicate  $d_2$  now).

$d_1(\square) \leftarrow \quad d_3(X) \leftarrow d_3(X)$   
 $d_1([X|X_s]) \leftarrow d_3(X), d_1(X_s)$

This gives an indication that always unfolding the parsing clause might be necessary to obtain a program that is fully equivalent to the original object program. On the other hand, if we aim at obtaining a large degree of specialisation, it might be worthwhile to stop unfolding at a parsing clause, especially when information propagation in the atoms of the object conjunction is involved. Consider the following program  $P$ :

**Example 3.12**

$$\begin{array}{lll} a(\text{mammal}) \leftarrow & b(\text{cat}, \text{mammal}) \leftarrow & p([X|X_s]) \leftarrow a(X), p(X_s) \\ a(X) \leftarrow b(X, Y), a(Y). & b(\text{dog}, \text{mammal}) \leftarrow & \\ & b(\text{eagle}, \text{bird}) \leftarrow & \end{array}$$

Using  $U_{\triangleleft}^+$ , partial deduction of  $\text{solve}(p(X))$  returns the following program  $P_1$ :

$$\begin{array}{lll} d_1([X|X_s]) \leftarrow d_2(X), d_1(X_s). & & \\ d_2(\text{mammal}) \leftarrow & d_2(\text{cat}) \leftarrow & d_2(\text{dog}) \leftarrow \end{array}$$

whereas always unfolding the parsing clause during specialisation of  $V_P$  would yield a program essentially equal to  $P$ . If  $V_P$  is specialised using  $U_{\triangleleft}$  without the extension  $U_{ext}$ , due to a generalisation, a less specialised program is obtained.

$$\begin{array}{ll} d_1(p([X|X_s]) \leftarrow d_3(X, X_s) & d_2(\text{cat}) \leftarrow \\ d_1(a(X)) \leftarrow d_2(X, Y) & d_2(\text{dog}) \leftarrow \\ d_1(a(\text{mammal})) \leftarrow & d_3(\text{mammal}, [X|X_s]) \leftarrow d_3(X, X_s) \\ d_1(b(\text{cat}, \text{mammal})) \leftarrow & d_3(X, [Y|Y_s]) \leftarrow d_2(X), d_3(Y, Y_s) \\ d_1(b(\text{dog}, \text{mammal})) \leftarrow & \\ d_1(b(\text{eagle}, \text{bird})) \leftarrow & \end{array}$$

So, contrary to what is generally believed, unfolding parsing calls during partial deduction does not always lead to optimal results. Unfolding some of the parsing calls, however, seems appropriate. This leaves us with several open issues, to which we will briefly return in Section 5.

## 4 Specialising An Extended Vanilla Meta Interpreter

In [7, 6], Brogi and Contiero discuss specialisation of the following extended vanilla meta interpreter, adapted here to our notation:

$$\begin{array}{l} \text{demo}(E, \text{true}) \leftarrow \\ \text{demo}(E, (A, B)) \leftarrow \text{demo}(E, A), \text{demo}(E, B). \\ \text{demo}(E, H) \leftarrow \text{clause}(E, H, B), \text{demo}(E, B). \\ \text{clause}(\text{union}(E_1, E_2), H, B) \leftarrow \text{clause}(E_1, H, B). \\ \text{clause}(\text{union}(E_1, E_2), H, B) \leftarrow \text{clause}(E_2, H, B). \\ \text{clause}(\text{inters}(E_1, E_2), H, B) \leftarrow \text{clause}(E_1, H, B), \text{clause}(E_2, H, B). \\ \text{clause}(\text{enc}(E), H, \text{true}) \leftarrow \text{demo}(E, H). \\ \text{clause}(\text{pr}(P), H, B) \leftarrow \text{statement}(P, H, B). \end{array}$$

These *clause* predicates implement the classic notion of operations needed for program composition: *union*, *intersection* and *encapsulation*. Single object programs, then, are represented by  $\text{statement}(P, H, B)$  facts, where  $P$  is the program name,  $H$  the clause head and  $B$  the clause body. See [8, 9, 5, 7] for an elaborated discussion of these composition operators.

In the following,  $D$  will denote the extended vanilla interpreter. The combination of  $D$  and a fully instantiated composition of object programs  $e$  will be denoted by  $D_e$ .

In [7], a specific program specialisation technique is developed in order to deal with these program compositions. A thorough discussion of the specialisation method is outside the scope of

this extended abstract, but it is noteworthy that the method relies heavily on knowledge about  $D$ . In case no *enc* operation is present in  $e$ , specialisation of  $D_e$  will lead to a program where all manipulations of  $e$  are removed. [7] also notice that this program *can* be transformed into an equivalent object program. No indication, however, is given how to automate this transformation, possibly also relying on information about  $D$ . In case  $e$  contains an *enc* operation, the method of [7] removes all explicit handling of  $e$ , but the resulting program is, in essence, a set of different *solve* vanilla meta interpreters, each with an associated *pclause* database. Thus no overhead due to the interpretation of a single program is removed. Also, since the object level goal is not taken into account during specialisation, no extra object specialisation can be achieved.

An interesting question is how our general, automatic, partial deduction technique handles specialisation of  $D_e$ . Instead of giving a full formal comparison with our method, we present some important observations. Concrete examples and experimental results can be found in Appendix C. As these examples show, our method is capable of achieving object specialisation as well.

**Theorem 4.1** For any program composition  $e$ ,  $PD(D_e, demo(e, X))$  will contain no meta structure concerning  $e$ .

**Theorem 4.2** For any program composition  $e$  not containing an *enc* operation,  $PD(D_e, demo(e, p(\overline{X})))$  with  $p$  a predicate in  $e$ , will contain no meta structure.

## 5 Discussion and Further Work

Although in this paper, we did not elaborate all details of the partial deduction method used, we would like to stress some important facts. If removal of structure is an important goal, it is necessary not to weaken the variant check on the global level. Also, the use of  $\triangleleft$  on *atoms* instead of *characteristic* atoms, is too weak. In some examples (e.g. Example 3.12), the use of characteristic atoms was a key factor in obtaining object specialisation. On the other hand, experiments indicated that the way in which a characteristic tree is imposed upon a generalisation [24, 25] should be refined through further unfolding the greatest common initial subtree. Always simply imposing the latter gives rise to unwanted *clause* predicates in the global tree.

The effect of always unfolding parsing calls, as described in [20], can be achieved by altering  $U_{\triangleleft}^+$ : instead of only using  $U_{ext}$  on atoms that would cause a generalisation when brought in the global tree,  $U_{ext}$  could be applied to *every* atom that is brought in the global tree. On the other hand, we demonstrated that keeping the object atoms together in one *solve* atom can cause more specialisation to be achieved. In the near future, we will investigate to what extent similar effects can be obtained through conjunctive partial deduction [23, 17]. Keeping the object atoms together in one *solve* atom often results in what we called *specialised* parsing. The question whether this is good or bad is not easily answered, since besides 'parsing', unifications on different head arguments are often involved, indicating that the performance of the specialised program can be system dependent.

As several experiments showed, generalising  $solve(C)$  with  $C$  a meta representation of an object conjunction, and further processing the generalised atom often does result in a node  $G$  on which  $solve(C)$  can be mapped during code generation without loss of meta structure. This indicates that  $U_{\triangleleft}^+$  is perhaps too rash, splitting *solve* atoms when it is not necessary and suggests that the decision for which atoms residual code will be generated, should be based on even more information, including information from the global level. This may require to (partially) redo some local unfoldings when more information gets available. However, this is a topic of further research.

$U_{ext}$  and  $U_{\triangleleft}^+$  as introduced in this paper are general rules, not requiring any information about what precisely is "meta" structure. The resulting partial deduction method nevertheless deals

very well with  $V$ , and is the first non ad hoc method to do so. In further work, we will investigate its performance in other contexts, including the specialisation of other, more involved meta interpreters.

## **Acknowledgements**

We thank Michael Leuschel for stimulating discussions, and for patiently explaining to us some of the inner workings of his ECCE partial deduction system. We also thank Danny De Schreye and Karel De Vlamincx for their interest and support. Finally, we are grateful to anonymous referees for providing valuable feedback which helped to improve the paper.

## References

- [1] J. Barklund. Metaprogramming in logic. In A. Kent and J.G. Williams, editors, *Encyclopedia of Computer Science and Technology*. Marcell Dekker, Inc., New York. To Appear.
- [2] K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, 1993.
- [3] R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
- [4] 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*. MIT Press, 1995.
- [5] A. Brogi and S. Contiero. Gödel as a meta language for composing logic programs. In A. Turini, editor, *Proceedings Meta'94*. University of Pisa, 1994.
- [6] A. Brogi and S. Contiero. A program specialiser for meta-level compositions of logic programs. Technical Report TR-96-20, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, May 1996.
- [7] A. Brogi and S. Contiero. Specialising meta-level compositions of logic programs. In J. Gallagher, editor, *Proceedings LOPSTR'96*, Stockholm, 1997. Springer-Verlag, LNCS 1207.
- [8] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition operators for logic theories. In J. W. Lloyd, editor, *Proceedings of the Esprit Symposium on Computational Logic*, pages 117–134. Springer-Verlag, November 1990.
- [9] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for modularising logic programming. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 105–119. Springer-Verlag, LNCS 649, 1992.
- [10] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.
- [11] D. De Schreye, M. Leuschel, and B. Martens. Tutorial on program specialisation (abstract). In J.W. Lloyd, editor, *Proceedings ILPS'95*, pages 615–616, Portland, Oregon, December 1995. MIT Press.
- [12] J. Gallagher. Transforming logic programs by specialising interpreters. In *Proceedings ECAI'86*, pages 109–122, 1986.
- [13] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, Computer Science Department, University of Bristol, U.K., November 1991.
- [14] J. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.
- [15] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 229–244, Leuven, April 1990.
- [16] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3&4):305–333, 1991.

- [17] R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S.D. Swierstra, editors, *Proceedings PLILP'96*, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag, LNCS 1140.
- [18] P. M. Hill and J. Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, U.K., August 1994. To appear in Volume V of the Handbook of Logic in Artificial Intelligence and Logic Programming, Oxford University Press.
- [19] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In H. D. Abramson and M. H. Rogers, editors, *Proceedings Meta'88*, pages 23–51. MIT Press, 1989.
- [20] A. Lakhotia and L. Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8(1):61–70, 1990.
- [21] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, *Proceedings of LOPSTR'95*, pages 1–16. Springer-Verlag, LNCS 1048, 1996.
- [22] M. Leuschel and D. De Schreye. An almost perfect abstraction operation for partial deduction using characteristic trees. Technical Report CW215, Departement Computerwetenschappen, K.U.Leuven, Belgium, October 1995. Submitted for Publication. Accessible via <http://www.cs.kuleuven.ac.be/~lpai>.
- [23] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.
- [24] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings Dagstuhl Seminar on Partial Evaluation*, pages 263–283, Schloss Dagstuhl, Germany, 1996. Springer-Verlag, LNCS 1110. Extended version as Technical Report CW220, K.U. Leuven. Accessible via <http://www.cs.kuleuven.ac.be/~lpai>.
- [25] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. Technical Report CW248, Departement Computerwetenschappen, K.U.Leuven, Belgium, February 1997. Accessible via <http://www.cs.kuleuven.ac.be/publicaties/rapporten/CW1997.html>.
- [26] Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via <http://www.cs.kuleuven.ac.be/~lpai>, 1996.
- [27] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3&4):217–242, 1991.
- [28] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, Departement Computerwetenschappen, K.U.Leuven, Belgium, February 1994.
- [29] 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.

- [30] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 28(2):89–146, 1996. Abridged and revised version of Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, October 1993, accessible via <http://www.cs.kuleuven.ac.be/~lpai>.
- [31] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Kanagawa, Japan, June 1995. MIT Press.
- [32] S. Safra and E. Shapiro. Meta interpreters for real. In H.-J. Kugler, editor, *Information Processing 86*, pages 271–278, 1986.
- [33] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
- [34] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Proceedings ILPS'95*, pages 465–479, Portland, Oregon, December 1995. MIT Press.
- [35] L. Sterling and R. D. Beer. Meta interpreters for expert system construction. *Journal of Logic Programming*, 6(1&2):163–178, 1989.
- [36] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.
- [37] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to metaprogramming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.

## A Controlling Automatic Partial Deduction

In recent years, considerable progress has been achieved on the issue of controlling automated partial deduction. In that context, a clear conceptual distinction was introduced between local and global control [14, 31].

The former deals with the construction of (possibly incomplete) SLD-trees for the atoms to be partially deduced. In essence, it consists of an unfolding strategy. Requirements are: termination, good specialisation, avoiding search space explosion as well as work duplication. Approaches have been based on one or more of the following elements:

- determinacy [16, 13]  
Only (except once) select atoms that match a single clause head. The strategy can be refined with a so-called “look-ahead” to detect failure at a deeper level. Methods solely based on this heuristic, apart from not guaranteeing termination, tend not to worsen a program, but are often somewhat too conservative.
- well-founded measures [10, 30]  
Imposing some (essentially) well-founded order on selected atoms guarantees termination, but, on its own, can lead to overly eager unfolding.
- homeomorphic embedding [34, 24]  
Instead of well-founded ones, well-quasi-orders can be used [33, 3]. Homeomorphic embedding on selected atoms has recently gained popularity as the basis for such an order.

At the global control level, closedness [27] is ensured and the degree of polyvariance is decided: For which atoms should partial deductions be produced? Obviously, again, termination is an important issue, as well as obtaining a good overall specialisation. The following ingredients are important in recent approaches:

- characteristic trees [16, 13, 22, 21]  
A characteristic tree is an abstraction of an SLD-tree. It registers which atoms have been selected and which clauses were used for resolution. As such, it provides a good characterisation of the computation and specialisation connected with a certain atom (or goal). Its use in partial deduction lies in the control of polyvariance: Produce one specialised definition per characteristic tree encountered.
- global trees [31, 24]  
Partially deduced atoms (or characteristic atoms, see below) can be registered in a tree structure that is kept well-founded or well-quasi-ordered to ensure (global) termination. In general, doing so, while maintaining closedness, requires abstraction (generalisation).
- characteristic atoms [21, 24]  
Recent work has shown that the best control of polyvariance can be obtained not on the basis of either syntactical structure (atoms) or specialisation behaviour (characteristic trees) separately, but rather through a combination of both. Such pairs consisting of an atom and an associated (imposed) characteristic tree are called *characteristic atoms*.

Finally, subsidiary transformations, applicable in a post-processing phase, have been proposed, e.g. to remove certain superfluous structures [15, 2] or to reduce unnecessary polyvariance [24].

## B Proof sketches

We first introduce some notation: For any program  $P$ , we denote by  $\mathcal{L}_P$  the language underlying  $P$ . If  $e$  denotes a program composition, consisting of programs  $P_1, \dots, P_n$ , then  $\mathcal{L}_e$  denotes the language of  $e$ , i.e.  $\mathcal{L}_e = \mathcal{L}_{P_1} \cup \dots \cup \mathcal{L}_{P_n}$ .

The proof of Proposition 3.4 and Corollary 3.5 is straightforward, since *pclause* is non recursive. Moreover, Proposition 3.4 and Corollary 3.5 also hold when using  $U_{\triangleleft}^+$ .

**Lemma B.1** For any program  $P$ . Let  $A$  be an atom in  $\mathcal{L}_P$ . Let  $C$  be a meta representation of a conjunction of atoms in  $\mathcal{L}_P$ ;  $C = (A_1, (A_2, \dots, (A_{n-1}, A_n)))$ . Then  $U_{ext}(solve((A, C))) = \{solve(A), solve(C)\}$ .

**Proof**  $solve((A, C))$  can be deterministically<sup>5</sup> unfolded by using clause (2) of  $V$ , resulting in  $solve(A), solve(C)$ . Since during this unfolding no instantiations are made in  $A$  nor  $C$ ,  $solve(A) \triangleleft solve((A, C))$  and  $solve(C) \triangleleft solve((A, C))$  by Definition 2.2. Moreover, since only one unfolding was performed, the set  $\{solve(A), solve(C)\}$  is the first set  $\{p_1, \dots, p_n\}$  found such that  $\forall i \in \{1 \dots n\}$ :  $solve(p_i) \triangleleft solve((A, C))$ . So, by Definition 3.7,  $U_{ext}(solve((A, C))) = \{solve(A), solve(C)\}$ .  $\square$

The following two lemma's imply that every atom in  $\mathcal{G}(V_P, G)$  for a program  $P$  and goal  $G = solve(p(\overline{X}))$  with  $p$  a predicate in  $P$ , is of the form  $solve(A)$  with  $A$  an atom in  $\mathcal{L}_P$ .

**Lemma B.2** For any program  $P$ , goal  $G$ ,  $\forall G_0 \in \mathcal{G}(V_P, G)$ : If  $G_0 = solve((A, C))$  with  $A$  an atom in  $\mathcal{L}_P$  and  $C$  an atom or a meta representation of a conjunction of atoms in  $\mathcal{L}_P$ , then, using  $U_{\triangleleft}^+$  and global control as defined in Section 2,  $\exists G_a \in Anc(G_0)$  such that  $G_a \triangleleft G_0$ , where  $Anc(G_0)$  denotes the set of ancestors of  $G_0$  in  $\mathcal{G}(V_P, G)$ .

**Proof** Suppose  $\exists G_0, G_a \in \mathcal{G}(V_P, G)$  with  $G_a \in Anc(G_0)$  and  $G_0 = solve((A, C))$  such that  $G_a \triangleleft G_0$ . By Algorithm 3.8,  $U_{ext}(G_0) = G_0$ . Since  $G_0 = solve((A, C))$ , this contradicts Lemma B.1.  $\square$

**Lemma B.3** For any program  $P$ , goal  $G = solve(p(\overline{X}))$  with  $p$  a predicate in  $P$ , let  $\mathcal{G}(V_P, G)$  be the global tree built by  $U_{\triangleleft}^+$  and the global control as defined in Section 2. Then  $solve(true) \notin \mathcal{G}(V_P, G)$ .

**Proof**  $solve(true)$  is not the root of  $\mathcal{G}(V_P, G)$  and since  $U_{\triangleleft}$  will always unfold  $solve(true)$ , it cannot occur in a local leaf, and therefore not in  $\mathcal{G}(V_P, G)$ .  $\square$

The following lemma states that if two atoms in some  $\mathcal{G}(V_P, G)$  are generalised, the structure lost by the generalisation will not contain any meta structure.

**Lemma B.4** For any program  $P$ , goal  $G$ , let  $\mathcal{G}(V_P, G)$  be the global tree built by  $U_{\triangleleft}^+$  and the global control as defined in Section 2. For any  $G_0, G' \in \mathcal{G}(V_P, G)$ : If  $G_0 \triangleleft G'$  such that  $G_0$  is generalised to  $G_{gen} = msg(G_0, G')$ , then  $G_0 = G_{gen}\theta$  where  $\theta$  is a substitution not containing any meta structure.

### Proof

From Corollary 3.5, Lemma's B.2 and Lemma B.3,  $G_0 = solve(p(t_1, \dots, t_n))$  with  $p(t_1, \dots, t_n)$  an atom in  $\mathcal{L}_P$ . Since  $G_0 \triangleleft G'$ ,  $G' = solve(p(s_1, \dots, s_n))$  with  $p(s_1, \dots, s_n)$  an atom in  $\mathcal{L}_P$ . By the definition of *msg*,  $msg(G_0, G') = solve(p(g_1, \dots, g_n))$ , where  $\forall i \in \{1 \dots n\}$ :  $g_i = msg(t_i, s_i)$ , where  $t_i$  and  $g_i$  are terms in  $\mathcal{L}_P$ . From this, the result follows.  $\square$

From Lemma B.4, the proof of Theorem 3.10 is straightforward.

---

<sup>5</sup> using a look-ahead.

**Lemma B.5** For any  $D_e$ , atoms with *clause* or *statement* as predicate are always unfolded by  $U_{\triangleleft}$ .

**Proof**

- *statement* predicates are non recursive and therefore atoms with *statement* predicates are always unfolded by  $U_{\triangleleft}$ .
- *clause* predicates are recursive, but the first argument of either *clause* or *demo* is fully instantiated and, as can be seen from the definition of  $D$ , never growing between successive *clause* or *demo* calls. It even always shrinks in size between successive clause calls. Therefore, in an SLD-tree, no atom  $clause(e_1, t_1)$  can have an ancestor  $clause(e_2, t_2)$  such that  $clause(e_2, t_2) \triangleleft clause(e_1, t_1)$ .

□

**Corollary B.6** For any program composition  $e$  and goal  $G$ ,  $\mathcal{G}(D_e, G)$  contains only *demo* atoms if  $\mathcal{G}(D_e, G)$  is built using the techniques of Section 2 and  $U_{\triangleleft}$  (or  $U_{\triangleleft}^+$ ).

**Lemma B.7** Consider two atoms  $demo(e_1, t_1)$ ,  $demo(e_2, t_2)$ :  $demo(e_1, t_1)$  ancestor of  $demo(e_2, t_2)$  in  $\mathcal{G}(D_e, G)$ . If  $demo(e_1, t_1) \triangleleft demo(e_2, t_2)$ , then  $e_1 = e_2$ .

**Proof**

Suppose  $e_1 \neq e_2$ ; since  $e_1 \triangleleft e_2$ ,  $e_2$  should be strictly greater than  $e_1$ . But then, by the definition of  $D$ ,  $demo(e_2, t_2)$  cannot descend from  $demo(e_1, t_1)$ . □

**Lemma B.8** For a program composition  $e$  and atoms  $t_1, t_2$  in  $\mathcal{L}_e$ ,  $msg(demo(e, t_1), demo(e, t_2)) = demo(e, msg(t_1, t_2))$ .

**Proof** Follows from the definition of  $msg$ . □

**Proof of Theorem 4.1**

Since all atoms in  $\mathcal{G}(D_e, demo(e, X))$  have *demo* as predicate (Lemma B.5), all we must prove is that, when the new predicates are generated, the program composition is filtered away during code generation. The only reason why it would not be filtered away, is because of a generalisation. But by Lemmas B.7 and B.8,  $\forall G \in \mathcal{G}(D_e, demo(e, X))$ ,  $G = demo(e', t)$ ,  $e'$  will not be generalised. □

**Proof of Theorem 4.2**

If  $e$  contains no *enc* operation,  $\forall G \in \mathcal{G}(D_e, demo(e, p(\overline{X})))$ ,  $G = demo(e, t)$ . Since every atom in  $\mathcal{G}(D_e, demo(e, p(\overline{X})))$  contains the same  $e$ , and since this  $e$  does not influence the generalisations made (see Lemmas B.7 and B.8), this first argument from all nodes  $G$  can be deleted, and the building of  $\mathcal{G}(D_e, demo(e, p(\overline{X})))$  can be seen as equivalent to the building of  $\mathcal{G}(V_P, solve(p(\overline{X})))$  for some program  $P$ , and the proof of Theorem 3.10 can be used. □

At last, the proof of Theorem 3.9 is given:

**Proof of Theorem 3.9**

1. We will first prove termination of the algorithm.

Since  $\tau_{U_{\triangleleft}}(A)$  is a finite, possibly incomplete SLD-tree, it has a finite amount of leaves. Therefore, we only need to concentrate on one such leaf. For every branch  $\beta$  in an SLD-tree  $\tau$ ,

ending in a leaf  $L$ , extending this leaf, by repeatedly performing  $U_{ext}$ , will result in the branch  $\beta$  being lengthened a number of times into  $\beta'$ . The set of nodes  $\{G_0, \dots, G_n\} \in \beta'$  is defined as follows:  $G_0 = L$ ,  $\forall i \in \{1 \dots n\}$ :  $G_i = Leaf(U_{ext}(G_{i-1}))$ , where  $Leaf(D)$  denotes the end point of the derivation  $D$ . No other branches are created in  $\tau$  since  $U_{ext}$  only performs deterministic unfolding. Now, we will proof that  $\beta'$  has a finite number of nodes.

- $U_{ext}(A)$  creates a finite derivation  $\leftarrow A = K_0 \dots \leftarrow K_m$ , where  $\forall i \in \{0 \dots m\}$ ,  $K_i$  denotes a goal in the derivation. This follows from Definition 3.7 since  $\sqsubseteq$  is a well quasi order.
- Consider the nodes  $\{G_0 \dots G_n\}$ . This set is finite. Otherwise  $\exists i, j$  with  $i < j$ :  $s(G_i) \sqsubseteq s(G_j)$ . But, by Definition 3.7, condition (\*) and transitivity of  $\triangleleft$ ,  $s(G_j) \triangleleft s(G_i)$ , and we have a contradiction.

2. The second part of Theorem 3.9 states that the result of the algorithm is an SLD-tree.

By Definition 3.7, if  $U_{ext}(A) = \leftarrow G_0 \dots \leftarrow G_m$ , then  $\leftarrow G_0 \dots \leftarrow G_m$  is an SLD derivation. The operator  $\uplus_L$  extends an SLD-tree  $\tau$  by replacing a leaf  $L = \leftarrow A_1, \dots, A_{j-1}, G_0, A_{j+1}, \dots, A_n$  by the SLD-derivation  $\leftarrow A_1, \dots, A_{j-1}, G_0, A_{j+1}, \dots, A_n \dots \leftarrow A_1, \dots, A_{j-1}, G_m, A_{j+1}, \dots, A_n$ , resulting in a new SLD-tree  $\tau'$ .

□

## C Examples

In this appendix, we show some examples from [6]. Without giving an in depth comparison of the method of [6, 7] with our automatic method, we merely give the obtained results.

Consider the following example, adapted from [6]. The example consists of 3 program modules: *path*, *train* and *ic*.

```
statement(path, path(_C1, _C2), train(_C1, _C2, _T)).
statement(path, path(_C1, _C2), (train(_C1, _C3, _T), path(_C3, _C2))).

statement(train, train(florence, pisa, reg), true).
statement(train, train(florence, rome, int), true).
statement(train, train(pisa, genova, nat), true).
statement(train, train(pisa, rome, nat), true).
statement(train, train(milan, florence, int), true).
statement(train, train(milan, pisa, nat), true).

statement(ic, train(_X, _Y, int), true).
```

### C.1 Without an *enc* operation

#### Example C.1

$PD(D_e, demo(union(pr(path), inters(pr(train), pr(ic))), X1))$ ,  
where  $e$  denotes  $union(pr(path), inters(pr(train), pr(ic)))$ .

```
demo(union(pr(path), inters(pr(train), pr(ic))), X1) :-
    demo__1(X1).
demo__1(true).
demo__1(’, ’(X1, X2)) :-
    demo__1(X1),
    demo__1(X2).
demo__1(path(X1, X2)) :-
    demo__2(X1, X2).
demo__1(path(X1, X2)) :-
    demo__3(X1, X2).
demo__1(train(florence, rome, int)).
demo__1(train(milan, florence, int)).
demo__2(florence, rome).
demo__2(milan, florence).
demo__3(milan, rome).
```

Notice how `demo__1` performs top level parsing, which is unavoidable, since we started with an object goal that is unknown at specialisation time. Besides this top level parsing, all meta structure has vanished from the program. Notice how our automatic technique has performed some specialisation, unrelated to the handling of the program compositions, by precomputing all possible paths (`demo__2` and `demo__3`).

In order to avoid this top level parsing, let us partially evaluate the same example, but now w.r.t. a top level predicate `path(X, Y)`:

$PD(D_e, demo(union(pr(path), inters(pr(train), pr(ic))), path(X, Y)))$ :

```

demo(union(pr(path),inters(pr(train),pr(ic))),path(X1,X2)) :-
    demo__1(X1,X2).
demo__1(florence,rome).
demo__1(milan,florence).
demo__1(X1,X2):-
    demo__2(X1,X3), demo__1(X3,X2).
demo__2(florence,rome).
demo__2(milan,florence).

```

Notice how the threat of generalisation causes  $U_{\triangleleft}^+$  to split  $demo(e, (train(X, Z), path(Z, Y)))$  into  $demo(e, train(X, Z))$  and  $demo(e, path(Z, Y))$ , thereby loosing the information propagation<sup>6</sup>. In order to avoid this generalisation, let us partially evaluate the same example, but now w.r.t. a new top level predicate  $p$ .

```
statement(path,p(X,Y),path(X,Y)).
```

$PD(D_e, demo(union(pr(path),inters(pr(train),pr(ic))),p(X,Y)))^7$ :

```

demo(union(pr(path),inters(pr(train),pr(ic))),p(X1,X2)) :-
    demo__1(X1,X2).
demo__1(florence,rome).
demo__1(milan,florence).
demo__1(X1,X2) :-
    demo__2(X1,X2).
demo__2(milan,rome).

```

For comparison, the method of [6] yields the following program:

```

path(X,Y):-train(X,Y,T).
path(X,Y):-train(X,Z,T), path(Z,Y).
train(florence, rome, int).
train(milan, florence, int).

```

## C.2 With an *enc* operation

### Example C.2

$PD(D_e, demo(union(pr(path), enc(inters(pr(train), pr(ic))), X1))$ ,  
where  $e$  denotes  $union(pr(path), enc(inters(pr(train), pr(ic))))$ :

```

demo__1(true).
demo__1(', '(X1,X2)) :-
    demo__1(X1),
    demo__1(X2).
demo__1(path(X1,X2)) :-
    demo__2(X1,X2,X3).
demo__1(path(X1,X2)) :-
    demo__3(X1,X3,X4,X2).
demo__1(', '(X1,X2)) :-

```

<sup>6</sup>Using  $U_{\triangleleft}$  instead of  $U_{\triangleleft}^+$  leads to generalisation and a program similar to the one containing top level parsing.

<sup>7</sup>After deterministic post-unfolding, a new `demo__1(milan,rome)` fact replaces the `demo__2` predicate.

```

demo__4(X1),
demo__4(X2).
demo__1(train(florence,rome,int)).
demo__1(train(milan,florence,int)).
demo__2(florence,rome,int).
demo__2(milan,florence,int).
demo__3(milan,florence,int,rome).
demo__4(true).
demo__4(','(X1,X2)) :-
demo__4(X1),
demo__4(X2).
demo__4(train(florence,rome,int)).
demo__4(train(milan,florence,int)).

```

In this example, surrounding the intersection between *train* and *ic* with an encapsulation operation does not change the obtained degree of specialisation: besides the necessary top level parsing, no meta structure is present: all manipulations concerning the program compositions have been specialised away, as well as all manipulations concerning parsing of the single object programs. Moreover, all possible paths have once again been precomputed during the specialisation<sup>8</sup>.

Compare this result with the obtained result of the specific technique, as reported in [6]:

```

demo(union(pr(path), enc(inters(pr(train),pr(ic)))), X):- demo_0(X).
demo_0(_E, true).
demo_0(_E, (_H,_T)):-demo_0(_E, _H),demo_0(_E,_T).
demo_0(_E, _H):-clause_0(_E, _H, _B), demo_0(_E, _B).
clause_0(path(_C1, _C2), train(_C1, _C2, _T)).
clause_0(path(_C1, _C2), (train(_C1, _C3,_T), path(_C3, _C2, _T))).
clause_0(X, true):- demo_1(X).
demo_1(_E, true).
demo_1(_E, (_H,_T)):-demo_1(_E, _H),demo_1(_E,_T).
demo_1(_E, _H):-clause_1(_E, _H, _B), demo_1(_E, _B).
clause_1(train(florence, rome, int), true).
clause_1(train(milan, florence, int), true).

```

This example illustrates well the fact that, using the technique of [6, 7], all overhead due to the handling of the program compositions has vanished, but none of the overhead due to the handling (interpretation) of single object programs has.

### C.3 Object Specialisation

We conclude with an example, showing that extra information in the object goal can be used to obtain specialisation of the object program:

#### Example C.3

$PD(D_e, demo(union(pr(train),pr(path)),path(X,rome)))$ ,  
where  $e$  denotes  $union(pr(train),pr(path))$ :

---

<sup>8</sup>During top level parsing, the duplication of ',' is due to some strange behaviour of the *enc* composition operator: the original program, when interpreted by  $D$ , will return infinitely many times the same answer, due to the fact that  $demo(e, (A,B))$  unifies with clause (3) of  $D$ , what results in new *demo* atoms, instead of failing during *clause* (where such a situation would fail when using  $V$ ). Off course, this behaviour is reflected in the specialised program.

```

demo(union(pr(path),pr(train)),path(X1,rome)) :-
    demo__1(X1).
demo__1(florence).
demo__1(pisa).
demo__1(X1):-
    demo__2(X1,X2),demo__1(X2).
demo__2(florence,pisa).
demo__2(florence,rome).
demo__2(pisa,genova).
demo__2(pisa,rome).
demo__2(milan,florence).
demo__2(milan,pisa).

```

The same remark as in Example C.1 can be made. Specialisation w.r.t.  $p(X,rome)$  yields (after deterministic post-unfolding and removing duplicate clauses):

```

demo(union(pr(path),pr(train)),p(X1,rome)) :-
    demo__1(X1).
demo__1(florence).
demo__1(pisa).
demo__1(milan).

```