TLP **2** (4 & 5): 461–515, July & September 2002.
DOI: 10.1017/S147106840200145X Printed in the United Kingdom

Logic program specialisation through partial deduction: Control issues

MICHAEL LEUSCHEL

Department of Electronics & Computer Science, University of Southampton, Highfield SO17 1BJ, UK

MAURICE BRUYNOOGHE

Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Abstract

Program specialisation aims at improving the overall performance of programs by performing source to source transformations. A common approach within functional and logic programming, known respectively as partial evaluation and partial deduction, is to exploit partial knowledge about the input. It is achieved through a well-automated application of parts of the Burstall-Darlington unfold/fold transformation framework. The main challenge in developing systems is to design automatic control that ensures correctness, efficiency, and termination. This survey and tutorial presents the main developments in controlling partial deduction over the past 10 years and analyses their respective merits and shortcomings. It ends with an assessment of current achievements and sketches some remaining research challenges.

KEYWORDS: program specialisation, logic programming, partial evaluation, partial deduction

1 Introduction

Program specialisation aims at improving the overall performance of programs by performing source to source transformations. A common approach, known as *partial evaluation* is to guide the transformation by partial knowledge about the input. In contrast to ordinary evaluation, partial evaluation is processing a given program P along with only *part* of its input, called the *static input*. The remaining part of the input, called the *dynamic input*, will only be known at some later point in time (which we call *runtime*). Given the static input S, the partial evaluator then produces a *specialised* version P_S of P which, when given the dynamic input D, produces the same output as the original program P. The program P_S is also called the *residual program*.

The theoretical feasibility of this process, in the context of recursive functions, has already been established in Kleene (1952), and is known as Kleene's S-M-N theorem. However, while Kleene was concerned with theoretical issues of computability and his construction often yields functions which are more complex to evaluate than the

original, the goal of partial evaluation is to exploit the static input in order to derive more efficient programs.

Consider, for example, the following program written in some informal functional syntax, to compute the n-th power of a given value x, where both $x, n \in \mathbb{N}$.

Example 1

$$\begin{array}{lcl} power(x,n) & = & \textit{if } (n = 1) \textit{ then } x \\ & & \textit{else } (x * power(x, n-1)) \end{array}$$

Now, suppose we specialise the above program for the situation where we want to compute the fifth power, that is $n = 5$. Looking at the definition of the *power* function, we notice that the following statements depend only on the value of n:

- the test of conditional statement,
- the expression $n - 1$ in the recursive call,
- the recursive call, since the recursion is completely determined by the value of n.

Performing these statements, and residualising the others, the result of specialising the call *power(x,5)* is the residual program:

$$power(x, 5) \quad = \quad x * x * x * x * x$$

If the specialiser is correct, the residual program computes the same function as the original program, but naturally only for inputs of which the static part equals the values with respect to which the program was specialised. In the above example, the residual program $power(x, 5)$ still implements the *power* function, but only the *fifth*-power function. It can be used to compute the fifth power of any value, but can no longer compute the n-th power of a value.

As the example illustrates, a partial evaluator evaluates those parts of P which only depend on the static input S and generates code for those parts of P which require the dynamic input D. This process has therefore also been called *mixed computation* (Ershov, 1982). What distinguishes partial evaluation from other program specialisation approaches is that the transformation process is guided by the available input. Because part of the computation has already been performed beforehand by the partial evaluator, the hope that we obtain a more efficient program P_S seems justified.

Partial evaluation (Consel & Danvy, 1993; Jones *et al.*, 1993; Mogensen & Sestoft, 1997) has been applied to many programming languages: functional programming languages (e.g. Jones *et al.*, 1993), logic programming languages (e.g. Gallagher, 1993; Komorowski, 1992; Pettorossi & Proietti, 1994), functional logic programming languages (e.g. Alpuente *et al.*, 1996, 1998a, 1998b; Lafave & Gallagher, 1997), term rewriting systems (e.g. Bondorf, 1988, 1989) and imperative programming languages (e.g. Andersen, 1992, 1994).

In the context of logic programming, full input to a program P consists of a goal G and evaluation corresponds to constructing a complete SLDNF-tree for $P \cup \{G\}$. For partial evaluation, the static input takes the form of a goal G' which is more general (less instantiated) than a typical goal G at runtime. In contrast to other

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

programming languages, one can still execute P for G' and (try to) construct an SLDNF-tree for $P \cup \{G'\}$. So, at first sight, it seems that partial evaluation for logic programs is almost trivial and just corresponds to ordinary evaluation.

However, since G' is not yet fully instantiated, the SLDNF-tree for $P \cup \{G'\}$ is usually infinite and ordinary evaluation will not terminate. A technique which solves this problem is known under the name of *partial deduction*. Its general idea is to construct a finite number of finite trees and to extract from these trees a new program that allows any instance of the goal G' to be executed.

Overview. We present the essentials of this technique in section 2. Then, in section 3 we identify the main issues in controlling partial deduction, which we then address in much more detail in sections 4 and 5. In section 6 we then discuss so-called *conjunctive partial deduction*, which extends partial deduction in that it can specialise entire conjunctions instead of just atoms. Finally, in section 7 we discuss issues that arise for various extensions of logic programming and conclude with a critical evaluation of the practical applicability of existing partial deduction systems and techniques.

Terminology. The term "partial deduction" has been introduced in Komorowski (1992) to replace the term partial evaluation in the context of pure logic programs (no side effects, no cuts). Though in section 4.5 we briefly touch upon the consequences of the impure language constructs, we adhere to this terminology because the word "deduction" places emphasis on the purely logical nature of the source programs. Also, while partial evaluation of functional and imperative programs evaluates only those expressions which depend exclusively on the static input, in logic programming one can, as we have seen above, in principle also evaluate expressions which depend on the unknown dynamic input. This puts partial deduction closer to techniques such as *unfold/fold* program transformations (Burstall & Darlington, 1977; Pettorossi & Proietti, 1994), and therefore using a different denomination seems justified. Note that partial evaluation and in particular partial deduction is not limited to evaluation of expressions based on the static input. It can also exploit data present in the source code of the program or gathered though program analysis. Finally, in the remainder of this article we suppose familiarity with basic notions in logic programming (Apt, 1990; Lloyd, 1987).

2 Basics of partial deduction

In this section we present the technique of partial deduction, which originates from Komorowski (1982). Other introductions to partial deduction can be found in Komorowski (1992), Gallagher (1993) and Leuschel (1999b). Note that, for clarity's sake, we deviate slightly from the original formulation of Lloyd & Shepherdson (1991).

To avoid constructing infinite SLDNF-trees for partially instantiated goals, the technique of *partial deduction* is based on constructing finite, but possibly *incomplete* SLDNF-trees. The clauses of the specialised program are then extracted from these

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

trees by constructing one specialised clause per branch. A single resolution step with a specialised clause now corresponds to performing *all* the resolutions steps (using original program clauses) on the associated branch.

Before formalising the notion of partial deduction, we briefly recall some basics of logic programming (Apt, 1990; Lloyd, 1987). Syntactically, programs are built from an alphabet of variables (as usual in logic programming, variable names start with a capital), function symbols (including constants) and predicate symbols. Terms are inductively defined over the variables and the function symbols. Formulas of the form $p(t_1,\ldots,t_n)$ with p/n a predicate symbol of arity $n \geqslant 0$ and $t_1,\ldots,t_n$ terms are atoms. Literals come in two kinds; positive literals are simply atoms; negative literals are of the form *not* A with A an atom. A *definite clause* is of the form $a \leftarrow B$ where the head a is an atom and the body B is a conjunction of atoms. In *normal clauses*, the body B is a conjunction of literals. A formula of the form $\leftarrow B$ with B a conjunction of atoms is a *definite goal*, with B a conjunction of literals, it is a *normal goal*. Definite, respectively normal *programs* are sets composed of definite, respectively normal clauses. In analogy with terminology from other programming languages, a literal in a clause body or in a goal is sometimes referred to as a *call*.

As detailed in Apt (1990) and Lloyd (1987), a *derivation step* selects an atom in a definite goal according to some *selection rule*. Using a program clause, it first renames apart the program clause to avoid variable clashes and then computes a most general unifier (*mgu*) between the selected atom and the clause head and, if an *mgu* exists, derives the *resolvent*, a new definite goal. (We also say that the selected atom is *resolved* with the program clause.) Now, we are ready to introduce our notion of SLD-derivation. As common in works on partial deduction, it differs from the standard notion in logic programming theory by allowing a derivation that ends in a nonempty goal where no atom is selected.

Definition 1
Let P be a definite program and G a definite goal. An *SLD-derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G_0 = G$, G_1, ... of goals, a sequence C_1, C_2, ... of properly renamed clauses of P and a sequence θ_1, θ_2, ... of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

The initial goal of an SLD-derivation is also called the *query*. An SLD-derivation is a successful derivation or refutation if it ends in the empty clause, a failing derivation if it ends in a goal with a selected atom that does not unify with any properly renamed clause head, an incomplete derivation if it ends in a nonempty goal without selected atom; if none of these, it is an infinite derivation. In examples, to distinguish an incomplete derivation from a failing one, we will extend the sequence of a failing derivation with the atom **fail**. The totality of SLD-derivations form a search space. One way to organise this search space is to structure it in an SLD-tree. The root is the initial goal; the children of a (non-failing) node are the resolvents obtained by selecting an atom and performing all possible derivation steps (a process that we call the *unfolding* of the selected atom). Each branch of the tree represents an SLD-derivation. A *trivial* tree is a tree consisting of a single node – the root – without selected atom.

SLDNF-derivations and SLDNF-trees originates from the extension towards normal programs (Apt, 1990; Lloyd, 1987). As detailed in Apt (1990) and Lloyd (1987), a negative ground literal *not A* can be selected in a goal, in which case a *subsidiary* SLDNF-tree is built for the goal $\leftarrow A$. Eventually that tree contains a refutation in which case the original goal fails, or fails finitely in which case the original goal has a child – the resolvent – obtained by removing the negative literal (the mgu of this derivation step is the empty substitution). Although it is possible that a subsidiary tree never reaches the status where it contains a refutation or fails finitely, we will ignore that possibility for the time being, making the assumption that in such case the negative literal is not selected and the subsidiary tree is not created (all goals on branches extending the original goal will contain the negative literal). This assumption, that we reconsider in section 4.4, makes that the specialised program can be extracted from the *main* tree, the tree that starts from the initial goal. As a consequence, partial deduction for normal programs is hardly different from partial deduction for definite programs. Finally, we say that an SLDNF-tree (resp. SLDNF-derivation) is finite iff the main tree (resp. derivation) is finite. Observe that an SLDNF-tree can be finite (and its construction can terminate) while some of its subsidiary trees are infinite. Indeed, finding one successful derivation in an infinite subsidiary tree is sufficient to infer failure of the node containing the selected negative literal referred to by the subsidiary tree.

Note that *floundering*, the situation where it is impossible to select a literal in a goal because it consists solely of nonground negative literals, is only a special case of an incomplete derivation. In what follows, when we mention the branches of an SLDNF-tree, we mean the branches of the main tree.

We now examine how specialised clauses can be extracted from SLDNF-derivations and trees.

Definition 2

Let P be a program, $G = \leftarrow Q$ a goal, D a finite SLDNF-derivation of $P \cup \{G\}$ ending in $\leftarrow B$, and θ the composition of the *mgu*s in the derivation steps. Then the formula $Q\theta \leftarrow B$ is called the *resultant* of D.

Note that the formula is a clause when Q is a single atom, as is the case in standard partial deduction. *Conjunctive partial deduction* (Section 6) also allows Q to be a conjunction of several atoms. The relevant information to be extracted from an SLDNF-tree is the set of resolvents and the set of atoms occurring in the literals at the non-failing leaves.

Definition 3

Let P be a program, G a goal, and τ a finite SLDNF-tree for $P \cup \{G\}$. Let $D_1, \ldots, D_n$ be the non-failing SLDNF-derivations associated with the branches of τ. Then the *set of resultants*, *resultants*(τ), is the set whose elements are the resultants of $D_1, \ldots, D_n$ and the *set of leaves*, *leaves*(τ), is the set of atoms occurring in the final goals of $D_1, \ldots, D_n$.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

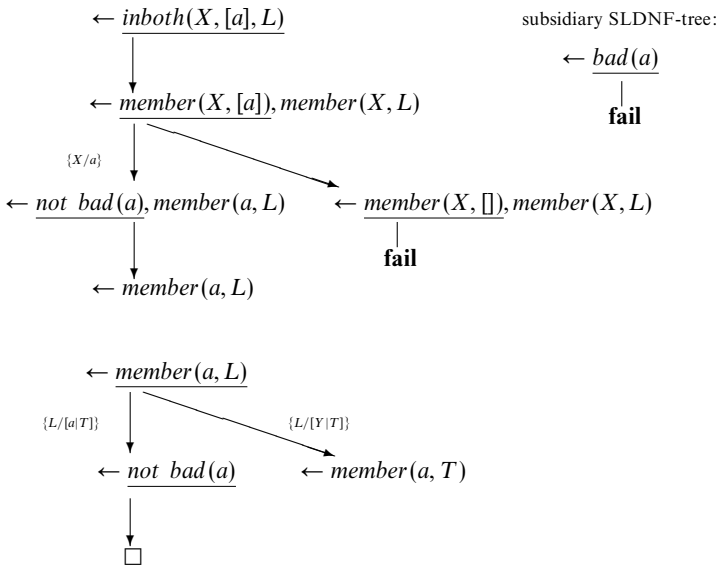

Fig. 1. Incomplete SLDNF-trees for Example 2.

Example 2
Let P be the following program:

$member(X, [X|T]) \leftarrow not\ bad(X)$
$member(X, [Y|T]) \leftarrow member(X, T)$
$inboth(X, L1, L2) \leftarrow member(X, L1), member(X, L2)$
$bad(b) \leftarrow$

Figure 1 represents an incomplete SLDNF-tree τ for $P \cup \{\leftarrow inboth(X, [a], L)\}$. This tree has just one non-failing branch and the set of resultants $resultants(\tau)$ contains the single clause:

$$inboth(a, [a], L) \leftarrow member(a, L)$$

Note that the complete SLDNF-tree for $P \cup \{\leftarrow inboth(X, [a], L)\}$ is infinite.

With the initial goal atomic, the extracted resultants are program clauses: the partial deduction of the atom.

Definition 4
Let P be a normal program, A an atom, and τ a finite non-trivial SLDNF-tree for $P \cup \{\leftarrow A\}$. Then the set of clauses $resultants(\tau)$ is called a *partial deduction of A in P*. If $\mathscr{A}$ is a finite set of atoms, then a *partial deduction of $\mathscr{A}$ in P* is the union of the sets obtained by taking one partial deduction for each atom in $\mathscr{A}$.

In analogy with terminology in partial evaluation, the partial deduction of A in P is also referred to as the *residual clauses* of A and the partial deduction of $\mathscr{A}$ in P as the *residual program*.

Example 3
Let us return to the program P of Example 2. Based on the trees in figure 1, we can construct the following partial deduction of $\mathscr{A} = \{inboth(X, [a], L), member(a, L)\}$ in P:

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

$member(a, [a|T]) \leftarrow$
$member(a, [Y|T]) \leftarrow member(a, T)$
$inboth(a, [a], L) \leftarrow member(a, L)$

Note that if τ is a trivial SLDNF-tree for $P \cup \{\leftarrow A\}$ then $resultants(\tau) = \{A \leftarrow A\}$ and the specialised program will be nonterminating for goals $\leftarrow A\theta$. The problem is avoided by excluding trivial trees in Definition 4.

The intuition underlying partial deduction is that a program P can be replaced by a partial deduction of $\mathscr{A}$ in P and that both programs are equivalent with respect to queries which are constructed from instances of atoms in $\mathscr{A}$. A first issue to clarify is what is intended by equivalent. Lloyd & Shepherdson (1991) where the first to examine it in detail. Using the completion semantics as the declarative semantics, they can only show soundness: that logical consequences from the completion of the specialised program are also logical consequences of the completion of the original program; the other direction, completeness (for instances of atoms in $\mathscr{A}$), does not hold in general, it holds only for programs for which SLDNF is a complete proof procedure. Note that the soundness result implies that answers obtained by SLDNF from the specialised program are sound with respect to the original program for any declarative semantics for which SLDNF is a sound procedure. For procedural equivalence under the SLDNF proof procedure, Lloyd and Shepherdson were able to obtain simple conditions guaranteeing equivalence. The correctness with respect to the well-founded semantics (now widely acknowledged to be better suited than completion semantics to capture the meaning of logic programs (Denecker *et al.*, 2001)) has been studied in Seki (1993), Przymusinska *et al.* (1994) and Aravindan & Dung (1994). The results allow us to conclude that partial deduction, as defined above, preserves declarative equivalence under the well-founded semantics for ground atoms that are instances of $\mathscr{A}$. Almost all works on partial deduction aim at preserving the procedural equivalence under SLDNF. Before defining the extra conditions required to ensure it, we introduce a few more concepts:

Definition 5
Let A_1, A_2, A_3 be three atoms, such that $A_3 = A_1\theta_1$ and $A_3 = A_2\theta_2$ for some substitutions θ_1 and θ_2. Then A_3 is called a *common instance* of A_1 and A_2. Let $\mathscr{A}$ be a finite set of atoms and S a set containing atoms, conjunctions, and clauses. Then S is *$\mathscr{A}$-closed* iff each atom in S is an instance of an atom in $\mathscr{A}$. Furthermore we say that $\mathscr{A}$ is *independent* iff no pair of atoms in $\mathscr{A}$ has a common instance.

The main result of (Lloyd & Shepherdson, 1991) about procedural equivalence can be formulated as follows:

Theorem 1 (*correctness of partial deduction*)
Let P be a normal program, $\mathscr{A}$ a finite, independent set of atoms, and P' a partial deduction of $\mathscr{A}$ in P. For every goal G such that $P' \cup \{G\}$ is $\mathscr{A}$-closed the following holds:

1. $P' \cup \{G\}$ has an SLDNF-refutation with computed answer θ iff $P \cup \{G\}$ does.
2. $P' \cup \{G\}$ has a finitely failed SLDNF-tree iff $P \cup \{G\}$ does.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

The theorem states that P and P' are procedurally equivalent with respect to the existence of success-nodes and associated answers for $\mathscr{A}$-closed goals. Furthermore, if we are in a setting where SLDNF is complete for a particular declarative semantics then partial deduction will preserve that semantics as well. Among others, this is the case for definite programs. For such programs the least Herbrand models of P and P' will have the same intersection with the set of $\mathscr{A}$-closed ground atoms. The fact that partial deduction preserves equivalence only for $\mathscr{A}$-closed goals distinguishes it from e.g. unfold/fold program transformations which aim at preserving equivalence for all goals. Note that the theorem does not tell us how to obtain $\mathscr{A}$. Also, it guarantees neither that termination, e.g. under Prolog execution, is preserved, nor that computed answers are found in the same order.

Returning to Example 3, we have that the partial deduction of the set $\mathscr{A} = \{inboth(X,[a],L),\ member(a,L)\}$ in P satisfies the conditions of Theorem 1 for the goals $\leftarrow inboth(X,[a],[b,a])$ and $\leftarrow inboth(X,[a],L)$ but not for the goal $\leftarrow inboth(X,[b],[b,a])$. Indeed, the latter goal succeeds in the original program but fails in the specialised one. Intuitively, if $P' \cup \{G\}$ is not $\mathscr{A}$-closed, then an SLDNF-derivation of $P' \cup \{G\}$ may select a literal for which no clauses exist in P' while clauses did exist in P. Hence, a query may fail while it succeeds in the original program, or, due to negation, may succeed while it fails in the original program. If $\mathscr{A}$ is not independent then a selected atom may be resolved with clauses originating from the partial deduction of two distinct atoms. This may lead to computed answers that, although correct, are not computed answers of the original program. Moreover, this can in turn lead to a specialised program that has a computed answer while the original program flounders. The next example illustrates these behaviours.

Example 4
Take the following program P:

$p(a,Y) \leftarrow q(Y)$
$p(X,b) \leftarrow$
$q(c) \leftarrow$

Let $\mathscr{A} = \{p(a,c)\}$. A partial deduction P' of $\mathscr{A}$ in P is:

$p(a,c) \leftarrow$

$P' \cup \{\leftarrow p(c,b)\}$ is not $\mathscr{A}$-closed and $P' \cup \{\leftarrow p(c,b)\}$ fails whereas $P \cup \{\leftarrow p(c,b)\}$ does not.
Now, let $\mathscr{A}' = \{p(a,X), p(Y,b)\}$. A partial deduction P'' of $\mathscr{A}'$ in P is:

$p(a,c) \leftarrow$
$p(a,b) \leftarrow$
$p(X,b) \leftarrow$

$\mathscr{A}'$ is not independent and $P'' \cup \{\leftarrow p(Z,b)\}$ produces the computed answers $\{Z/X\}$ and $\{Z/a\}$. The latter (redundant) answer is not produced by $P \cup \{\leftarrow p(Z,b)\}$. Moreover, $P'' \cup \{\leftarrow p(Z,b), \neg p(a,Z)\}$ produces the computed answer $\{Z/a\}$ whereas $P \cup \{\leftarrow p(Z,b), \neg p(a,Z)\}$ flounders. While one might consider this an improvement, it violates the requirement that the original and specialised program are procedurally equivalent for the goal.

Note that the original unspecialised program P is also a partial deduction of $\mathscr{A} =$

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

$\{member(X, L), inboth(X, L1, L2)\}$ in P, which furthermore satisfies the correctness conditions of Theorem 1 for any goal G. In fact, one can always obtain the original program back by putting into $\mathscr{A}$ an atom $p(X_1, \ldots, X_n)$ for every predicate symbol p of arity n and by constructing an SLDNF-tree of depth 1 for every atom in $\mathscr{A}$. In other words, neither Definition 4 nor the conditions of Theorem 1 ensure that any specialisation has actually been performed. Nor do they give any indication on how to construct a suitable set $\mathscr{A}$ and a suitable partial deduction wrt $\mathscr{A}$ satisfying the correctness criteria of the theorem. These considerations are all generally delegated to the *control* of partial deduction, which we discuss in detail in the following sections.

In the above development we deviated slightly from the original presentation in Lloyd & Shepherdson (1991). They define a *partial deduction of P wrt* $\mathscr{A}$ to be "a normal program obtained from P by replacing the set of clauses in P, whose head contains one of the predicate symbols appearing in $\mathscr{A}$ with a partial deduction of $\mathscr{A}$ in P." In other words, one keeps the original definitions for those predicates which do not appear in $\mathscr{A}$. Hence, Theorem 1 is a corollary of the results in Lloyd & Shepherdson (1991) and of the fact that the original definitions are not reachable from any call which is $\mathscr{A}$-closed. Note that our formulation, in contrast to Lloyd & Shepherdson (1991), thus enables partial deduction to eliminate dead code, i.e. code that can never be reached by executing a legal query to the specialised program. Hence, the original definition of (Lloyd and Shepherdson 1991) is not used in any partial deduction (or even partial evaluation) system we are aware of.

The following, more realistic example illustrates the practical benefits of partial deduction.

Example 5

Let us examine the following program, defining the higher-order predicate *map*, which maps predicates over lists:

$map(P, [], []) \leftarrow$
$map(P, [X|T], [Px|Pt]) \leftarrow C = ..[P, X, Px], call(C), map(P, T, Pt)$
$inv(0, 1) \leftarrow$
$inv(1, 0) \leftarrow$

Note that the above program can be seen as a pure definite logic program by conceptually adding a clause $call(p(X_1, \ldots, X_n)) \leftarrow p(X_1, \ldots, X_n)$ for each n-ary predicate symbol p and by adding a fact $= ..(f(X_1, \ldots, X_n), [f, X_1, \ldots, X_n])$ for each n-ary function symbol f.

If we now want to map the *inv* predicate on a list, then we can specialise the set $\mathscr{A} = \{map(inv, In, Out)\}$. If we build the incomplete SLDNF-tree represented in Figure 2, the set of all the leaf atoms is $\mathscr{A}$-closed and we can construct the following residual program:

$map(inv, [], []) \leftarrow$
$map(inv, [0|T], [1|Pt]) \leftarrow map(inv, T, Pt)$
$map(inv, [1|T], [0|Pt]) \leftarrow map(inv, T, Pt)$

All the higher-order overhead (i.e. the use of $= ..$ and *call*) has been removed; also the calls to $inv/2$ have been unfolded. When running the above programs on a set of queries one notices that the specialised program runs up to 2 times faster

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

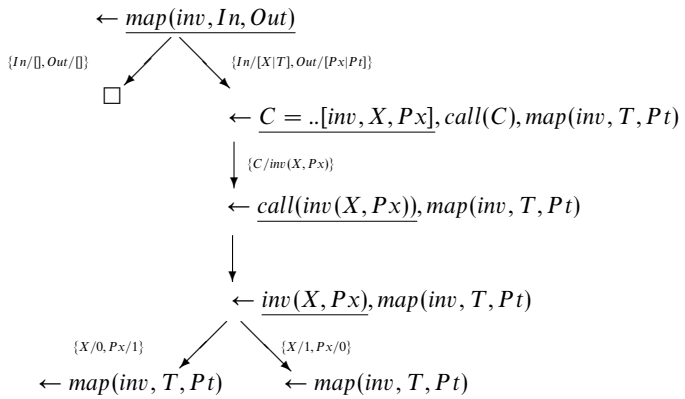

Fig. 2. Unfolding Example 5.

than the original one (depending on the particular Prolog system used; and can be made even faster using filtering, as discussed in section 5.1).

The question that remains is, how do we come up with such (non-trivial and correct) partial deductions in an automatic way? This is exactly the issue that is tackled in the remainder of this article.

3 Main control issues

Partial deduction starts from an initial set of atoms $\mathscr{A}$ provided by the user that is chosen in such a way that all runtime queries of interest are $\mathscr{A}$-closed. As we have seen, constructing a specialised program requires to construct an SLDNF-tree for each atom in $\mathscr{A}$. Moreover, one can easily imagine that the conditions for correctness formulated in Theorem 1 may require to revise the set $\mathscr{A}$. Hence, when controlling partial deduction, it is natural to separate the control into two components (as already pointed out in Gallagher (1993) and Martens & Gallagher (1995)):

- The *local control* controls the construction of the finite SLDNF-tree for each atom in $\mathscr{A}$ and thus determines *what* the residual clauses for the atoms in $\mathscr{A}$ are.
- The *global control* controls the content of $\mathscr{A}$, it decides *which* atoms are ultimately partially deduced (taking care that $\mathscr{A}$ remains closed for the initial atoms provided by the user).

This gives rise to the generic scheme for a partial deduction procedure (similar to the scheme in Gallagher (1993, 1995)) in figure 3.

The local control is exhibited by the function $unfold(P, A_k)$ that returns a finite SLDNF-tree for $P \cup \{\leftarrow A_k\}$. Once all trees constructed, the atoms in their leaves are added to the set of atoms. Then the global control, exhibited by the function $revise(\mathscr{A}'_i)$ is responsible for adapting the set of atoms in such a way that all atoms in $\mathscr{A}'_i$ (and thus S as well as all the leaves) are $\mathscr{A}_{i+1}$-closed and that, eventually,

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

Procedure 1
Input: A program P and a set S of atoms of interest;
Output: A specialised program P' and a set of atoms $\mathscr{A}$;
Initialise: $i = 0$, $\mathscr{A}_0 = S$;
repeat
for each $A_k \in \mathscr{A}_i$ **do**
let $\tau_k := unfold(P, A_k)$;
let $\mathscr{A}'_i := \mathscr{A}_i \cup \{B | B \in leaves(\tau_k)\}$;
let $\mathscr{A}_{i+1} := revise(\mathscr{A}'_i)$;
let $i := i + 1$
until $\mathscr{A}_i = \mathscr{A}_{i-1}$;
let $\mathscr{A} := \mathscr{A}_i$;
let $P' := \bigcup_{A_k \in \mathscr{A}} resultants(\tau_k)$

Fig. 3. Generic partial deduction procedure.

a fixpoint is reached where $\mathscr{A}_i = \mathscr{A}_{i-1}$ and a correct specialised program can be extracted. The specialised program can then be used for all queries that are $\mathscr{A}$-closed.

To turn this scheme into a correct and usable algorithm, several issues have to be considered. On the one hand, the specialised program has to be correct and the partial deduction has to terminate. On the other hand, the specialised program should be as efficient as feasible; it means that the available information, whether in the input or in the context of calls to predicates, has to be exploited as much as possible. These somewhat conflicting issues are elaborated below:

1. *Correctness.* It requires that the specialised program computes the same results as the original for queries that are $\mathscr{A}$-closed. Partial correctness is obtained by ensuring that Theorem 1 can be applied. This can be divided into a (very simple) local condition, requiring the construction of non-trivial trees, and into a global one related to the independence and closedness conditions.
2. *Termination.* There are two sources of potential nontermination. First, one has to ensure that a finite SLDNF-tree is generated in finite time. This is referred to as the *local* termination problem. Secondly, one has to ensure that the iteration over the successive sets $\mathscr{A}_i$ terminates and that the set itself remains finite (otherwise an infinite set of trees would have to be built). This is referred to as the *global* termination problem. A related pragmatic aspect is that the partial deduction process finishes in a reasonable amount of time. What is reasonable depends on the application, e.g. whether the specialised program is to be used once or many times; whether the partial deduction process is part of standard compilation or a separate process initiated by the user.
3. *Degree of specialisation.* The degree to which the available information is exploited is called the *degree of specialisation* or *precision*, and unexploited information is referred to as *precision loss*. We can again discern two aspects. One which we might call *local specialisation*. At first glance, the more atoms are unfolded, the more derivation steps are replaced by a single derivation step in the specialised program, hence the better the specialised program is. However, as discussed in section 4.1, one can unfold too much. Another issue related to local specialisation is that the atoms in a leaf of an SLDNF-tree are

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

treated separately. No information is exchanged between the SLDNF-trees of distinct atoms. For instance, if we stop the unfolding process in Example 2 for $G = \leftarrow inboth(X,[a,b,c],[c,d,e])$ at the goal $G' = \leftarrow member(X,[a,b,c]), member(X,[c,d,e])$, partial deduction will not be able to infer the fact that the only possible answer for G' and G is $\{X/c\}$ as the atoms $member(X,[a,b,c])$ and $member(X,[c,d,e])$ are specialised separately. (This problem is partially remedied by conjunctive partial deduction, c.f. section 6.) Continuing the unfolding of $G' = \leftarrow member(X,[a,b,c]), member(X,[c,d,e])$ achieves information propagation between the individual atoms and brings this fact to the surface, resulting in much better specialisation.

The second aspect could be called the *global specialisation* and is related to the granularity of $\mathscr{A}$. In general having a more precise and fine grained set $\mathscr{A}$ (with more *instantiated* atoms) will lead to better specialisation. For instance, given the set $\mathscr{A} = \{member(a,[a,b]), member(c,[d])\}$, partial deduction can perform much more specialisation (i.e. detecting that the goal $\leftarrow member(a,[a,b])$ always succeeds exactly once and that $\leftarrow member(c,[d])$ fails) than given the less instantiated set $\mathscr{A}' = \{member(X,[Y|T])\}$, where $member(X,[Y|T])$ is the most specific atom which is more general than the atoms in $\mathscr{A}$.

A third aspect, orthogonal to both previous ones, is the size of the specialised program. Unfolding too much may result in code explosion, huge specialised programs, not only requiring lots of memory but perhaps also slowing down the execution. What counts for the user is not the amount of unfolding but the speed of the specialised program. Unfortunately, the actual performance is hard to predict and hence is not used to guide the specialisation process in current approaches.

4 Local control

The function $unfold(P,A)$, introduced in the generic partial deduction procedure of Section 3, that computes a finite SLDNF-tree for $P \cup \{\leftarrow A\}$ encapsulates the local control and implements what is called an *unfolding strategy*. The unfolding strategy performs a finite number of derivation steps, starting from the query $\leftarrow A$. It should not be confused with the unfold rule in the unfold/fold program transformation framework that performs a single derivation step on an atom selected in a clause body.

The unfolding strategy applied on an atom A determines exactly the SLDNF-tree for that atom, hence its residual clauses. Consequently, it has a big impact on the efficiency of the final program. In the next section, we explain why too much unfolding can lead to inefficient residual clauses and how such deterioration can be prevented.

4.1 Efficiency by determinacy

Example 6

The well known append program is as follows:

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

$app([], L, L) \leftarrow$
$app([H|X], Y, [H|Z]) \leftarrow app(X, Y, Z)$

Now, let us try to specialise this program without having any partial input, i.e. $\mathscr{A} = \{app(X, Y, Z)\}$. If we build an SLDNF-tree of depth 1 for $app(X, Y, Z)$ we just get the original program back. We have not obtained any improvements, but at least we have not worsened the program either. Actually, without any partial input, this is the best we can do. Indeed, if we unfold more and, for example, perform two unfolding steps we obtain the following residual program:

$app([], L, L) \leftarrow$
$app([X], L, [X|L]) \leftarrow$
$app([H, H'|X], Y, [H, H'|Z]) \leftarrow app(X, Y, Z)$

Although the residual program performs only half of the resolution steps performed by the original program, it is not more efficient on standard Prolog implementations. Indeed, the code size has increased and the resolution steps themselves have become more complicated. Performing more unfolding steps makes things worse, as the following table shows (we ran a set of typical queries using SICStus Prolog 3.8.6 on a Linux'86 machine; relative runtimes are actual runtimes divided by runtime of the original program):

Unfolding depth	1	2	3	4	5	6	7	8	9	10	11	12
Relative runtime	1	1.3	1.6	1.6	1.7	1.8	1.9	2.0	2.0	2.2	2.4	2.5

As the table shows, two extra unfolding steps already incur a performance penalty of 60%. This illustrates that too much unfolding can seriously harm the efficiency of the residual program. The result of such transformations may well be very implementation dependent as not only unifications are more complex but also the clause selection process. The overhead of the latter is dependent on the quality of the indexing of the implementation. As the phenomenon is typical for cases where the number of clauses increases, one could call it *local code explosion* (there is a similar problem of code explosion at the global level when the set $\mathscr{A}$ gets too large).

Another pitfall of too much unfolding is known as *work duplication*. The problem is illustrated in the following example.

Example 7

Let P be the following program (adapted from Example 2):

$member(X, [X|T]) \leftarrow$
$member(X, [Y|T]) \leftarrow member(X, T)$
$inboth(X, L1, L2) \leftarrow member(X, L1), member(X, L2)$

Let $\mathscr{A} = \{inboth(a, L, [X, Y]),\ member(a, L)\}$. By performing the non-leftmost non-determinate unfolding for $inboth(a, L, [X, Y])$ in Figure 4 (and doing the same unfolding for $member(a, L)$ as in figure 1), we obtain the following partial deduction P' of P with respect to $\mathscr{A}$:

$member(a, [a|T]) \leftarrow$
$member(a, [Y|T]) \leftarrow member(a, T)$
$inboth(a, L, [a, Y]) \leftarrow member(a, L)$
$inboth(a, L, [X, a]) \leftarrow member(a, L)$

Let us examine the run-time goal $G = \leftarrow inboth(a, [h, g, f, e, d, c, b, a], [X, Y])$, for which $P' \cup \{G\}$ is $\mathscr{A}$-closed. Using the Prolog left-to-right computation rule the

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

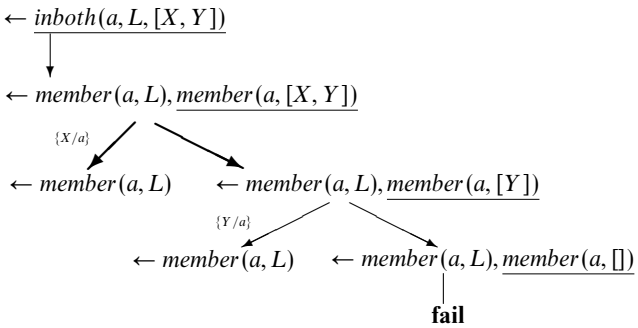

Fig. 4. Non-leftmost non-determinate unfolding for Example 7.

expensive sub-goal $\leftarrow member(a, [h, g, f, e, d, c, b, a])$ is only evaluated once in the original program P, while it is executed twice in the specialised program P'.

Observe that this is not a problem of local code explosion as in Example 6. The increase from one to two *inboth*/3 clauses is arguably normal as calls to *member*/2 have been unfolded and this predicate is defined by two clauses.

Some partial evaluators, for instance, SAGE (Gurr, 1994b, a994a) do not prevent such work duplication. This can result in arbitrarily big slowdowns, much higher than those encountered in Example 6 (e.g. see Bowers & Gurr, 1995).

A common approach to prevent local code explosion and work duplication relies on determinacy-based unfolding. It was first proposed in Gallagher & Bruynooghe (1991) and Gallagher (1991, 1993).

Definition 6
The unfold function is *determinate* iff for every program P and every goal G it returns an SLDNF-tree with at most one non-failing branch.

Applying determinate unfolding to an atom will produce an SLDNF-tree with at most one resultant. Hence no local code explosion and no work duplication can occur. Also, determinacy is a strong indication that enough input is available to select the "right" derivation, the derivation that will be taken when the specialised program is executed for the dynamic input.

Finally, determinate unfolding ensures that the order of solutions, e.g. under Prolog execution, is not altered and that termination is preserved (termination might however be improved, as e.g. $\leftarrow loop, fail$ can be transformed into $\leftarrow fail$; for further details related to the preservation of termination we refer to Proietti & Pettorossi (1991), Bossi & Cocco (1994, 1995) and Leuschel *et al.* (1998b)).

It is undecidable whether, for a given literal, one can construct an SLDNF-tree with at most one non-failing branch. Hence, concrete unfold functions use a so-called *lookahead* to decide whether a particular literal can be unfolded. Using a lookahead of 0 means that a literal can only be unfolded if it produces one resultant or less, while using a lookahead of 1 means that we can also select literals which produce more than one resultant, provided that all but one of them fail at the next resolution step.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

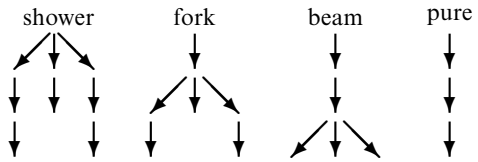

Fig. 5. Three almost determinate trees and one determinate tree.

The determinate unfolding approach is too restrictive, as we have to prevent trivial trees, and is usually replaced by *almost determinate unfolding* that allows one non-determinate unfolding step. This non-determinate step may either occur only at the root (used, for example, in Gallagher (1991)), only at the bottom (used in Gallagher & Bruynooghe (1991) and Leuschel & De Schreye (1998a)), or anywhere in the tree (an option which can be used within ECCE (Leuschel, 1996)). These three forms of *almost determinate* trees are illustrated in figure 5. However, as the experiments in Leuschel *et al.* (1998a) show, even almost determinate unfolding can be too restrictive and does not fare very well on highly non-deterministic programs, such as the "contains" benchmark (Leuschel 1996) devised by Lam and Kusalik. Nonetheless, as we will see in section 6, this is much less of an issue in the setting of so-called conjunctive partial deduction.

To avoid the work duplication pitfall described in Example 7, the one non-determinate unfolding step performed by an almost determinate unfolding rule should mimic the runtime selection rule (leftmost for Prolog). Observe that for a shower tree this is always satisfied, as there is only one literal in the root.

Among the three almost determinate unfolding trees, the shower is the most restrictive one as it only allows a non-determinate step if necessary to avoid a trivial tree. All three avoid local code explosion as the number of residual clauses cannot exceed the number of program clauses defining the atom selected at the non-deterministic step.

Unfortunately, fork and beam determinate unfolding can still lead to duplication of work, namely in unification with multiple heads:

Example 8
Let us adapt Example 7 by using $\mathscr{A} = \{inboth(X, [Y], [V, W])\}$. We can fully unfold $\leftarrow inboth(X, [Y], [V, W])$ and we then obtain the following partial deduction P' of P with respect to $\mathscr{A}$:

$inboth(X, [X], [X, W]) \leftarrow$
$inboth(X, [X], [V, X]) \leftarrow$

No goal has been duplicated by the leftmost non-determinate unfolding, but the unification $X = Y$ for $\leftarrow inboth(X, [Y], [V, W])$ has been duplicated in the residual code. This unification can have a substantial cost when the corresponding actual terms are large. In fact, code like the above could as well be written by hand, and the problem could be attributed to poor compiler technology. We are here touching upon a rather low level issue on the borderline between specialisation and compilation that is not well mastered and not much studied. Ideally, unfolding decisions should be based on a more precise performance model that takes into

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

account the compiler technology of the target system such as clause indexing, the cost of term construction operations, and the cost of having too many arguments (often considerable slowdown occurs if the number of arguments exceed 32). In the absence of such detailed modelling and of better compiler technology, pragmatic solutions are either to use shower determinate unfolding only, or to provide a postprocessor that avoids the unification overhead through the introduction of explicit disjunctions (denoted ";" as in Prolog):

$$inboth(X, [X], [V, W]) \leftarrow (X = V) \ ; \ (X = W)$$

or, even better on most Prolog systems[1], through the introduction of an auxiliary predicate (so called transformational indexing):

$$inboth(X, [X], [V, W]) \leftarrow one\_of(X, V, W)$$
$$one\_of(X, X, \_) \leftarrow$$
$$one\_of(X, \_, X) \leftarrow$$

4.2 Ensuring termination

Having solved the problems of local code explosion and work duplication, we still have no adequate unfolding function. Indeed almost determinate unfolding can result in infinite branches. In (strict) functional programs such a condition is equivalent to an error in the original program. In logic programming (and in lazy functional programming) the situation is somewhat different: a goal can infinitely fail (in a deterministic way) during partial deduction but still finitely fail at run time, i.e. when executed using fully instantiated input. In applications where one searches an infinite space for the existence of a solution (e.g. theorem proving) even infinite failures (i.e. infinite SLDNF-trees without a refutation in the main tree) at run-time do not necessarily indicate an error in the program: they might simply be due to non-existence of a solution. This is why, perhaps in contrast with functional programming, additional measures on top of determinacy should be adopted to ensure local termination.

Early approaches either did not guarantee termination or made ad-hoc decisions to enforce termination. Subsumption checking (unfolding stops when the selected atom is an instance of a previously selected atom) and variant checking (unfolding stops when the selected atom is a variant of a previously selected atom) are examples of the former approach and are mentioned in Takeuchi & Furukawa (1986), Fuller & Abramsky (1988), Levi & Sardu (1988), Benkerimi & Lloyd (1990) and van Harmelen (1989), but are inadequate (Bruynooghe *et al.* (1992), as the following examples illustrate.

Example 9

Take the following simple program for reversing a list.

$$rev([], Acc, Acc) \leftarrow$$
$$rev([H|T], Acc, Res) \leftarrow rev(T, [H|Acc], Res)$$

[1] Private communication from Bart Demoen.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

Unfolding $\leftarrow rev(X, [], R)$ using subsumption or variant checking will give rise to an infinite SLD-tree.

The use of an arbitrary depth bound is an example of an ad-hoc approach. Unavoidably, there are cases where this leads to either too much unfolding and code explosion, or too little unfolding and under utilisation of the available information. The hope is that the other components of the unfolding strategy will cause that the depth bound is used only in pathological cases. Approaches using depth bounds are in Venken (1984), Prestwich (1993), Fuller *et al.* (1996) and Sahlin (1993, 1991).

4.2.1 Offline approaches

One approach to ensure termination is to perform a preliminary analysis and to use the results of this analysis to make the unfolding decisions.

1. *Offline annotations.* In this approach, often referred to as *offline* (because almost all the control decisions are taken before the actual specialisation phase), unfolding proceeds in a strict left-to-right fashion and every call in the program to be specialised has an annotation specifying whether it is to be unfolded or not. In the latter case the call is said to be residualised. One could annotate the programs by hand and then check whether the annotation is correct, i.e. the unfolding will terminate. This can be achieved by removing the literals annotated as to be residualised (as they are residualised, they are not executed and do not create bindings) and to use existing tools for termination analysis of logic programs (see De Schreye & Decorte (1994) for a survey and the specialised literature for more recent work). It is a component of the approach of Vanhoof & Bruynooghe (2001) described at the end of the next paragraph.

However, in general one also wants to automatically derive the annotations itself: this preliminary analysis is referred to as a *Binding-Time Analysis* (BTA). The first fully implemented bta for logic programs was probably presented in (Gurr, 1994a), for the SAGE system. This bta is monovariant and unfolding decisions are taken at the predicate level, i.e. for each predicate all calls are either unfolded or residualised. This is thus still too restrictive in practice. A more recent and more powerful bta (for functional programs), which ensures termination and can even handle sophisticated programs such as interpreters, is presented in Glenstrup & Jones (1996). Bruynooghe *et al.* (1998) presented a step towards a polyvariant bta for logic programs. Assuming an unfolding condition for every predicate is given, it employs abstract interpretation to derive a polyvariant version of the original program where every call is annotated with an unfolding decision (for some predicates, the clauses defining them can be multiplied and each version is differently annotated). Vanhoof & Bruynooghe (1999) have developed a binding time analysis for Mercury (Somogyi *et al.*, 1996), a typed and moded logic programming language. Given the features of Mercury, this work is closer to work in partial evaluation of functional programs than to partial deduction of logic programs. Vanhoof & Bruynooghe (1999) extended it to cope with the higher-order features and module structure of Mercury. Finally, Vanhoof

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

& Bruynooghe (1996) describe a full binding time analysis for logic programs. The termination analyser of Codish & Taboch (1999) has been extended for the case that it cannot prove termination. The extension identifies the atoms in clause bodies that are at the origin of the failure to prove termination. This termination analyser is then used in an iterative process. When it proves termination, all calls are annotated as unfoldable. In the other case, one of the identified atoms is annotated as to be residualised and the program with the residualised atom removed is again analysed for termination. Eventually, enough atoms are annotated as residualised to allow a proof that the execution (unfolding) terminates.

One of the big advantages of the offline approach is the efficiency of the specialisation process itself: indeed, once the annotations have actually been derived (automatically by the above btas or by hand), the specialiser is relatively simple, and can be made to be very efficient, since all decisions concerning local control are made before and not during specialisation.

The simplicity of the specialiser also means that it is much easier to achieve *self-application*, i.e. specialise the specialiser itself using partial evaluation. Indeed, achieving effective self-application was one of the initial motivations for investigating offline control techniques (Jones *et al*., 1989). Self-application was first achieved in the logic programming context in Mogensen & Bondorf (1992) for a subset of Prolog and later in Gurr (1994b, 1994a) for full Gödel. Self-application enables a partial evaluator to generate so-called "compilers" from interpreters using the second Futamura projection and a compiler generator (*cogen*) using the third Futamura projection (e.g. see Jones *et al*., 1993). However, the actual creation of the *cogen* according to the third Futamura projection is not of much interest to users since *cogen* can be generated once and for all when a specialiser is given. This is known as the *cogen-approach* and has been successfully applied in many programming paradigms (Beckman *et al*., 1976; Romanenko, 1988; Holst, 1989; Holst & Launchbury, 1992; Birkedal & Welinder, 1994; Andersen, 1994). In the logic programming setting, (Neumann, 1990, 1991) presents a system for definite clause grammars which is very similar to a *cogen*, but not from a partial evaluation perspective. The first *cogen* for a logic programming language was thus (arguably) presented in Jørgensen & Leuschel (1996) and Leuschel & Jørgensen (1999). The resulting system LOGEN performs the unfolding at speeds similar to ordinary execution, and is thus well suited for applications, where speed of the specialisation is crucial (and where the program to be specialised can be analysed beforehand by the bta).

2. *Delay declarations.* Instead of taking all unfolding decisions at analysis time, one can also infer conditions under which unfolding is guaranteed to terminate and leave it to the specialiser to check whether a particular atom meets the condition and can be unfolded. The specialiser, knowing the actual static input, may then be able to unfold more atoms than a binding time analyser would consider safe. The required analysis has lots in common with the analysis used for logic programs with *delay declarations* (also called coroutining). When executing such programs, calls are suspended until they meet their delay declarations. Analysis can be developed that can verify whether the program terminates for a given delay declaration or

that can infer delay declarations ensuring termination. Relevant work is in Naish (1993), Lüttringhaus-Kappel (1993), Marchiori & Teusink (1995) and Martin & King (1997). Using the delay declarations for which the program terminates to decide whether atoms should be unfolded or residualised ensures termination of unfolding (Incomplete branches in the SLDNF-tree correspond to deadlocked derivations).

Such an approach has actually not been very widely used yet, with the exception of Fujita & Furukawa (1988), Leuschel (1994), Leuschel & De Schreye (1998b) and Martin & Leuschel (1999) and Martin (2000). Note that some of the delay declarations derived by Naish (1992), Marchiori & Teusink (1995) and Martin & King (1997) can be overly restrictive in the context of unbounded (i.e. partially instantiated) datastructures (common in partial deduction). Hence, Martin & Leuschel (1999) and Martin (2000) extend this approach by pre-computing minimum sizes for the unbounded structures and unfold atoms as long as sizes remain under the minimum.

4.2.2 Online approaches: well-founded and well-quasi orders

In this section we look at so called online approaches that monitor the growth of branches of SLDNF-trees, continue unfolding as long as there is some evidence that interesting computations are performed but are also guaranteed to terminate. To achieve this, they maintain orders over the nodes of a branch that are chosen in such a way that infinite branches are impossible. If care is taken that there cannot be an infinite number of attempts to rebuild a branch, the construction of the tree must terminate.

Well-founded orders and well-quasi orders are well known to allow the definition of admissible sequences that are always finite. Their definitions are as follows:

Definition 7
A *strict partial order* $<_S$ on a set S is an irreflexive, transitive, and thus asymmetric binary relation on S. A *quasi order* (also called preorder) $\leqslant_S$ on a set S is a reflexive and transitive binary relation on S.

Definition 8
Let $<_S$ be a strict partial order on a set S. A sequence of elements $s_1, s_2, \ldots$ in S is called *admissible with respect to* $<_S$ iff $s_i + 1 < s_i$, for all $i \geqslant 1$. The relation $<_S$ is a *well-founded order* (*wfo*) iff there is no infinite admissible sequence with respect to $<_S$.

Definition 9
Let $\leqslant_S$ be a binary relation on S. A sequence of elements $s_1, s_2, \ldots$ in S is called *admissible with respect to* $\leqslant_S$ iff there are no $i < j$ such that $s_i \leqslant_S s_j$. The relation $\leqslant_S$ is a *well-binary relation* (*wbr*) on S iff there are no infinite admissible sequences with respect to $\leqslant_S$. The relation $\leqslant_S$ is a *well-quasi order* (*wqo*) on S iff it is a well-binary relation and a quasi order.

In what follows, we define an *expression* to be either a term, an atom, a conjunction, or a goal.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

When defining orders over the sequence of nodes in a branch, nobody has found it useful to compare complete goals, only the selected atoms are compared. Also, it was quickly realised that it was difficult to define an order relation on the full sequence that was giving good unfoldings and that it was sufficient and easier to do so on certain subsequences. The essence of the most advanced technique, based on covering ancestors Bruynooghe *et al.* (1992) can be captured in the following definitions.

Definition 10
If a program clause $H \leftarrow B_1, \ldots, B_n$ is used in a derivation step with selected atom A then, for each i, A is the *parent* of the instance of B_i in the resolvent and in each subsequent goal where an instance originating from B_i appears (up to and including the goal where B_i is selected). The *ancestor* relation is the transitive closure of the parent relation.

Definition 11
Let $G_0, G_1, \ldots, G_n$ be an SLDNF-derivation with selected atoms $A_1, A_2, \ldots, A_n$.

The *covering ancestor sequence of* A_i, a selected atom, is the maximal subsequence $A_{j_1}, A_{j_2}, \ldots A_{j_m} = A_i$ of $A_1, A_2, \ldots, A_i$ such that all atoms in the sequence have the same predicate symbol and, for all $1 \leqslant k < m$ it holds that A_{j_k} is an ancestor of $A_{j_{k+1}}$.

An SLDNF-derivation $G_0, G_1, \ldots, G_n$ is *safe with respect to an order* (wfo or wqo) if all covering ancestor sequences of the selected atoms are admissible with respect to that order.

Covering ancestors, first introduced for well-founded orders (Bruynooghe *et al.* (1992), and later also used with well-quasi orders (Leuschel *et al.*, 1998a), are so useful because an infinite derivation must have at least one infinite covering ancestor sequence. Hence, an atom can be unfolded when the SLDNF-derivation remains safe. Moreover, experience has shown that the admissibility of the covering ancestor sequences is a strong indication that some interesting specialisation is going on.

Well-founded orders. Inspired by their usefulness in the context of static termination analysis (see Dershowitz & Manna (1979) and De Schreye & Decorte (1994)), well-founded orders have been successfully employed to ensure termination of partial deduction in (Bruynooghe *et al.* (1992), Martens *et al.* (1994), Martens & De Schreye (1996) and Martens (1994). In addition, the unfolding performed by these techniques is related to the structural aspect of the program and goal to be partially deduced. They are arguably the first theoretically and practically satisfying solutions for the local termination problem.

Example 10
A simple well-founded order can be obtained by comparing the *termsize* of atoms: we say that $A < B$ iff $termsize(A) < termsize(B)$, where $termsize(t)$ of an expression t is the number of function and constant symbols in t. Let us apply this to the *member* program P of Example 2. Based on that wfo, the SLDNF-tree with successive goals $\leftarrow member(X, [a, b|T])$, $\leftarrow member(X, [b|T])$ and $\leftarrow member(X, T)$ results in

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

the covering ancestor sequence $member(X, [a, b|T])$, $member(X, [b|T])$, $member(X, T)$ which is admissible because the termsize of the selected atoms strictly decreases at each step. However, it is not allowed to perform a further unfolding step as the addition of the element $member(X, T')$ to the covering ancestor sequence makes the sequence inadmissible.

In general, measuring just the termsize of atoms leads to overly conservative unfolding. Take for example the *rev* program from Example 9. Given, e.g., the goal $\leftarrow rev([a, b], [], R)$ one would ideally want to achieve full unfolding. Fully unfolding $\leftarrow rev([a, b], [], R)$ results in a covering ancestor sequence $rev([a, b], [], R)$, $rev([b], [a], R)$, $rev([], [b, a], R)$. Unfortunately, as the termsize is 6 for all the elements, the sequence is not admissible and the derivation is not safe. However, using a wfo which just examines the termsize of the first argument, the branch is admissible and full unfolding can be achieved. This illustrates that it is difficult to decide beforehand which is the wfo that gives the best unfolding and that there is a need to adjust the wfo while unfolding.

Such an approach is followed in (Bruynooghe *et al.* (1992), Martens *et al.* (1994), Martens & De Schreye (1996) and Martens (1994). They start off with a simple wfo and then refine it during the unfolding process.

Example 11

Consider a query $G_1 = \leftarrow rev([a, b|T], [], R)$ for the *rev* program P of Example 9. One starts with the wfo based on summing up the termsizes of the arguments whose positions are in the set $S_1 = \{1, 2, 3\}$. Unfolding one step, the resolvent is $G_2 = \leftarrow rev([b|T], [a], R)$ and the covering ancestor sequence is $rev([a, b|T], [], R)$, $rev([b|T], [a], R)$. Using the wfo based on S_1, both atoms have size 5 and the covering ancestor sequence is inadmissible. The adjustment of the wfo removes a minimal number of elements from S_1 such that the sequence becomes admissible. Using $S_2 = \{1, 3\}$ achieves this. Another unfolding step yields the goal $G_3 = \leftarrow rev(T, [b, a], R)$ and the covering ancestor sequence remains admissible. Performing another unfolding step results in the goal $\leftarrow rev(T', [H', b, a], R)$ and the covering ancestor sequence $rev([a, b|T], [], R)$, $rev([b|T], [a], R)$, $rev(T, [b, a], R)$, $rev(T', [H', b, a], R)$, which is *not* admissible for S_2 and for any subset of it. Hence it is not allowed to perform the last step.

The above example suggests two critical points. First, one has to ensure that one cannot continuously refine a wfo. In the above example this was ensured by only allowing arguments to be removed. In a more general setting (e.g. where one can vary weights associated with constants and function symbols) one has to ensure that the successive wfos are themselves well-founded.

Secondly, when selecting a new wfo, verifying that the last atom in the covering ancestor sequence is strictly smaller than the previous one does not guarantee that the whole sequence is admissible (while it suffices when extending an admissible sequence for a given wfo with one atom). Hence, early algorithms tested the whole sequence on admissibility. This can be expensive for long sequences.

Martens & De Schreye (1996) and Martens (1994) therefore advocates another solution: not re-checking the entire sequence on the grounds that it does not threaten

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

termination (provided that the refinements of the wfo themselves are well-founded). This leads to sequences $s_1, s_2, \ldots$ of selected literals which are not well-founded but *nearly-founded* Martens & De Schreye (1996) and Martens (1994) meaning that $s_i \not< s_j$ only for a finite number of pairs (i, j) with $i > j$. This improves the efficiency of the unfolding process, but has the tradeoff that it can lead to sequences of covering ancestors which contain more than one occurrence of exactly the same selected literal (Leuschel, 1998a), which is considered a clear sign of too much unfolding.

Well-quasi orders. A drawback of the above mentioned wfo approaches, is that they will not be able to satisfactorily handle certain programs. For example, Datalog programs (logic programs without functors) will pose problems as all constants have the same size under the measures that are typically used in wfos. Assigning a different size to each constant will not solve the problem. As the ordering is total, there will be situations where it leads to suboptimal unfolding. For Datalog program on could use variant checking as the number of distinct variants is finite. A more fundamental solution is to use quasi orderings.

Local termination is ensured in a similar manner as for wfos by allowing only safe SLDNF-trees. The difference is that the admissibility of covering ancestor sequences is based on well-quasi orders. Hence an element added to an admissible sequence is not necessarily strictly smaller than all elements in the sequence as is the case for a wfo. This, for example, allows a wqo to have no *a priori* fixed size or order attached to functors and arguments and avoids to focus in advance on specific sub-terms. The latter is crucial to obtain good unfolding of metainterpreters (Leuschel, 1998b, 1998a).

The first explicit uses of wqos to ensure termination of partial deduction are in Bol (1993) and Sahlin (1993). Prestwich (1992a) presents a method which can be seen as a simple wqo: it maps atoms to so-called "patterns" (of which there are only finitely many) and unfolds every pattern at most once. Prestwich (1992a) also presents an improvement whereby it is always allowed to decrease the *termsize*. This can still be seen as a wqo. In fact, every wfo can be mimicked by a wqo and the combination of two wqos is still a wqo (Leuschel, 1998b, 1998a).

An interesting wqo is the homeomorphic embedding relation $\trianglelefteq$, which derives from results by Higman (1952) and Kruskal (1960). It has been used in the context of term rewriting systems in Dershowitz (1987) and Dershowitz & Jouannaud (1990), and adapted for use in supercompilation in Sørensen & Glück (1995).

What follows is an adaptation of the definition from Sørensen & Glück (1995), in turn based on the so-called pure $\trianglelefteq$ in Dershowitz & Jouannaud (1990). It has a simple treatment of variables.

Definition 12

The *homeomorphic embedding* relation $\trianglelefteq$ on terms and atoms is defined inductively as follows (i.e. $\trianglelefteq$ is the least relation satisfying the rules), where $n \geqslant 0$, p denotes predicate symbols, f denotes function symbols, and $s, s_1, \ldots, s_n, t, t_1, \ldots, t_n$ denote terms:

1. $X \trianglelefteq Y$ for all variables X, Y

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

2. $s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i
3. $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $\forall i \in \{1, \ldots, n\} : s_i \trianglelefteq t_i$.
4. $p(s_1, \ldots, s_n) \trianglelefteq p(t_1, \ldots, t_n)$ if $\forall i \in \{1, \ldots, n\} : s_i \trianglelefteq t_i$.

When $s \trianglelefteq t$ we also say that s is *embedded in* t or t is *embedding* s. By $s \triangleleft t$ we denote that $s \trianglelefteq t$ and $t \not\trianglelefteq s$. The important property is that $\trianglelefteq$ is a well-quasi order Sørensen & Glück (1995).

The intuition behind the above definition is that $A \trianglelefteq B$ iff A can be obtained from B by removing some symbols i.e. that the structure of A, splitted in parts, reappears within B. For instance we have $p(a) \trianglelefteq p(f(a))$ because $p(a)$ can be obtained from $p(f(a))$ by removal of "$f()$" Observe that the removal corresponds to the application of rule 2 (also called the diving rule) and that we also have $p(a) \triangleleft p(f(a))$. Other examples are $X \trianglelefteq X$, $p(X) \triangleleft p(f(Y))$, $p(X, X) \trianglelefteq p(X, Y)$ and $p(X, Y) \trianglelefteq p(X, X)$.

In order to adequately handle some built-ins, the embedding relation $\trianglelefteq$ of Definition 12 has to be adapted. Indeed, some built-ins (like $= ../2$ or $is/2$) can be used to dynamically construct new constants and functors. With an unbounded number of constants and functors, $\trianglelefteq$ is not a wqo. To remedy this Leuschel *et al.* (1998a) partition the constants and functors into the *static* ones (those occurring in the original program and the partial deduction query) and the *dynamic* ones (those created during program execution)[2]. As with the set of variables, the set of dynamic constants and functors is unbounded. Hence, not surprisingly a wqo is obtained by adding to Definition 12 a rule similar to the rule for variables:

$f(s_1, \ldots, s_m) \trianglelefteq g(t_1, \ldots, t_n)$ if both f and g are dynamic

Comparing wfos and wqos. The homeomorphic embedding allows us to continue unfolding in situations where no suitable wfo exists. For example, on its own (i.e. not superimposed on a determinate unfolding strategy) it will allow the complete unfolding of most terminating Datalog programs.

The homeomorphic embedding $\trianglelefteq$ allows also better unfolding in the context of metaprogramming (see Leuschel (1998a) and Vanhoof (2001)).

Take, for example, the atoms $A = p([], [a])$ and $B = p([a], [])$. This is a situation where a homeomorphic embedding allows more unfolding than any wfo: it allows us to unfold A when B is in its covering ancestor sequence, but also the other way around, i.e. it allows us to unfold B when A is in its covering ancestor sequence. A wfo will at best assign a different size to both atoms and the total order, fixed in advance implies that only one of the two unfoldings can be performed. The dynamic adjustment of wfos which we described in Example 11 will allow both unfoldings. However, if we make the above example slightly more complicated, e.g. by using the atoms $A = solve(p([], [a]))$ $B = solve(p([a], []))$ or even $A = solve_1(\ldots solve_n(p([], [a]))\ldots)$ $B = solve_1(\ldots solve_n(p([a], []))\ldots)$ instead, then the scheme of Example 11 will no longer work (while $\trianglelefteq$ still allows both unfoldings). For such a wfo scheme to allow both unfoldings, we have to make the dynamic argument selection process more refined but then we run into the problem that infinitely many dynamic refinements

[2] A similar division was used in MIXTUS (Sahlin, 1993) to solve problems with subsumption checking.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

might exists (Martens & De Schreye, 1996; Martens, 1994), and to our knowledge no satisfactory solutions exists as of yet.

However, the above example also illustrates why, when using a wqo, one has to compare with *every predecessor*. Otherwise one will get infinite derivations where in turn the atoms $p([a],[])$, $p([],[a])$ and again $p([a],[])$ are selected. When using a wfo one has to compare only to the closest predecessor, because of the transitivity of the order and the strict decrease enforced at each step.

Formally, one can prove that $\trianglelefteq$ is strictly more powerful than so-called *simplification orderings* (such as lexicographic path ordering; see Dershowitz & Jouannaud (1990)) and so-called *monotonic* wfos (Leuschel, 1998b): the admissible sequences with respect to $\trianglelefteq$ are a strict superset of the union of all admissible sequences with respect to simplification orderings and monotonic wfos. Almost all wfos presented in the online partial deduction literature so far fall into this category. Also, compared to all these wfo-approaches, the $\trianglelefteq$ approach is relatively easy to implement. The combined power and simplicity explains its popularity in the recent years (Sørensen & Glück, 1995; Leischel *et al.*, 1998a; Glück *et al.*, 1996; Jørgensen *et al.*, 1996; Alpuente *et al.*, 1997; Lafave & Gallagher, 1997; Albert *et al.*, 1998; De Schreye *et al.*, 1999).

There are, however, natural wfos which are neither simplification orderings nor monotonic. For such wfos, there can be sequences which are not admissible wrt $\trianglelefteq$ but which are admissible wrt the wfo. Indeed, $\trianglelefteq$ takes the whole term structure into account while wfos in general can ignore part of the term structure. For example, the sequence $\langle [1,2], [[1,2]] \rangle$ is admissible wrt the "listlength" measure but not wrt $\trianglelefteq$, where "listlength" measures a term as 0 if it is not a list and by the number of elements in the list if it is a list (Martens & De Schreye (1996).

In summary, the only circumstances when one might consider using wfos for online control instead of a wqo such as $\trianglelefteq$ are:

1. When the use of the wqo $\trianglelefteq$ is considered too inefficient (checking the extension of an admissible sequence for admissibility is much less expensive with a wfo than with a wqo).
2. When there is a need to consider only parts of the terms structures inside atoms. It is unclear how often this is going to be important in practice.
3. When one wants to explicitly restrict the amount of unfolding, e.g. for pragmatic reasons.

4.3 Local control in ECCE

Experience with ECCE, an *online* partial deduction system (Leuschel 1996), has resulted in the following recommendations for unfolding a goal: (the query is always unfolded, as needed for correctness):

- If the goal fails (has a literal that does not unify with any clause head) then label the derivation as a failing one.
- Else, try to find a determinate literal whose unfolding yields an SLDNF-derivation that is safe with respect to the wqo $\trianglelefteq$ and unfold it. To decide whether a literal is determinate a lookahead of 1 is used.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

- Else, unfold the leftmost literal and stop with further unfolding of this branch (apart from identifying failing resolvents). This rule is not always giving the best unfolding. There are derivations where non-determinate unfolding is better omitted. Also it can be that the leftmost literal is a built-in or another literal that cannot be unfolded because its definition is not available. In such case, non-leftmost non-determinate unfolding can be considered if the amount of work duplication to be introduced is minimal (which is the case for cheap built-ins such as $\backslash =$) or will be minimised by a postprocessor or smart Prolog compiler.

These recommendations are not always sufficient. On benchmarks such as the highly non-deterministic "contains" referred to in section 4.1, they are too restrictive. Obtaining good specialisation requires to perform non-determinate unfolding (and, as for determinate unfolding, it must be safe with respect to the wqo $\trianglelefteq$). Interestingly, the default setting of ECCE includes so-called "conjunctive" partial deduction (to be discussed in section 6) and determinate unfolding is sufficient to handle "contains" and similar benchmarks. The first version of ECCE described in Leuschel *et al.* (1998a) did not include conjunctive partial deduction and thus non-determinate unfolding was employed.

4.4 Termination within subsidiary SLDNF-trees

In an SLDNF-derivation, there is not only the possibility of non-termination for the main SLDNF-tree but also for all the subsidiary SLDNF-trees. Under SLDNF, such subsidiary trees are only created for ground atoms, hence their unfolding at specialisation-time is not different from their execution at run-time. However, as control is different, some subsidiary trees can be created during partial deduction which are never created at run-time. Moreover, the original program may be erroneous in the sense that the execution of some of the subsidiary trees created at run-time does not terminate. So, to ensure that the partial deduction of a program always terminates, one has to control the execution of the subsidiary trees.

Non-termination can have two sources. On the one hand, an infinite branch can be created. This is similar to the problem of creating an infinite branch in the main tree, and the same local control techniques can be used to prevent it. On the other hand, a ground negative literal can be selected in a subsidiary tree, leading to the creation of another subsidiary tree, and so on, eventually resulting into an infinite set of subsidiary trees. This problem is similar to the global termination problem mentioned in section 3 and can also be solved by the same techniques (to be described in section 5). Alternatively, one could conceptually attach the subsidiary trees to the main tree (i.e. when building a subsidiary tree for an atom A we consider all childrens of A also as childrens of $\neg A$ in the main tree) and then use the local control techniques which we discussed.

If the control interrupts the execution of the subsidiary tree before it reports

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

success or failure to the main node, then the negative atom cannot be selected and the node becomes either an incomplete leaf or another atom has to be selected.[3]

4.5 From pure logic programming to Prolog

Pure Prolog. As already mentioned, Theorem 1 guarantees neither that termination under, e.g. Prolog's left-to-right selection rule is preserved, nor that solutions are found in the same order. However, as shown in Proietti & Pettorossi (1991), there are further restrictions on the unfolding that can be imposed to remedy this (and no further restrictions on the global control are necessary). First, we have already seen that determinate unfolding can only improve termination and will not change the order of solutions under Prolog. Secondly, leftmost unfolding (determinate or not) changes neither the termination nor the order of solution under Prolog execution. Thus, if one prevents non-leftmost, non-determinate unfolding (as already discussed in Example 7 this is also a good idea for efficiency) then partial deduction will always preserve termination (and could improve it) as well as the order of solutions for pure Prolog programs.

Full Prolog. So far we have only considered pure logic programs with declarative built-ins (such as *functor*, *arg*, *call*, cf., Example 5). We were thus able to exploit the *independence of the selection rule* (Apt, 1990; Lloyd (1987), in the sense that the unfolding rule did not have to systematically select the leftmost literal in a goal. We were thus able, e.g., to perform non-leftmost determinate unfolding steps (which can be the source of big speedups, see Leuschel & De Schreye (1998b)). In this section we briefly touch upon the differences between partial deduction of pure logic programs and partial evaluation of impure Prolog.

When we move towards full Prolog with extra logical built-ins, such as *var*, the cut, or even *assert*, we can no longer make use of the independence of the selection rule and our unfolding choices become more limited as everything that modifies the procedural semantics of the program may have an effect on the results computed by it.

For the cut, the order of solutions is important, as the cut commits to the first solution. Predicates such as $nonvar/1$ and $var/1$ are what is called *binding-sensitive*. Success or failure for e.g. $var(X), p(X)$ can be different than for $p(X), var(X)$ and unfolding $p(X)$ in $var(X), p(X)$ can result in so called *backpropagation* of bindings onto the binding-sensitive call to $var/1$. Also the side effect of a printing statement is binding-sensitive and backpropagation of a failure may eliminate its execution altogether as in the specialisation of $print(hello), fail$ into $fail$. Thus, any non-leftmost unfolding step, even when determinate, may cause a change in the procedural semantics. Proposals to overcome this limitation can be found in, for example, O'Keefe (1985), Bugliesi & Russo (1989), Prestwich (1992b), Sahlin (1993, 1991) and Leuschel (1994). In essence, one has to avoid backpropagation of bindings onto

[3] In both cases the negative literal will feature in the residual program, and one should not throw the subsidiary trees away, as they can be used for code generation.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

binding-sensitive predicates. For example, given a program P containing a single fact $p(a) \leftarrow$ for the predicate p, the goal $\leftarrow var(X), q(X), p(X)$ (with q not binding-sensitive) is specialised into $\leftarrow var(X), X = a, q(a)$. This avoids the backpropagation of a into $var(X)$.

Similarly, one has to avoid backpropagation of failure onto predicates with side-effects such as *print*. For example, for the same program P and a goal $\leftarrow print(a), q(b)$, assuming all unfoldings of $q(b)$ end in failure, one cannot specialise the goal into $\leftarrow fail$ but has to specialise it into $\leftarrow print(a), fail$ instead.

A problem related to the cut is that unfolding an atom with a program clause containing a cut modifies the scope of the cut: the SLDNF-tree resulting from the execution of the specialised program is pruned differently by the cut than the SLDNF-tree from the execution of the original program. This problem is overcome by providing special built-ins (mark-cut). They allow us to preserve the meaning of cut under unfolding. The if-then-else, with its local cut, poses much less problems and is preferable from a partial evaluation perspective (O'Keefe, 1985).

Another problem relates to the specialisation of modules. Some systems (e.g. ECCE (Leuschel 1996)) allow some predicates to be annotated as *open*. The specialiser assumes that the definitions will be provided at runtime and does not unfold such predicates. (For specialising Prolog, one should in addition declare whether or not these predicates are binding-sensitive). A solution for the Gödel module system is presented in Gurr (1994a), using the concept of a *script* where the module structure has basically been flattened.

In summary, extending the control techniques to full Prolog is feasible. In essence, one has to prevent the backpropagation of bindings, either by only performing leftmost unfolding or by some other means (e.g. the explicit introduction of equalities). However, as backpropagation can lead to early detection of failure and hence important speedups, it means that some interesting specialisations are no longer possible. Figuring out, via some analysis, when a substitution can safely be back-propagated beyond a binding sensitive predicate call is a difficult challenge, and, to our knowledge, no satisfactory solution exists.

5 Global control

5.1 Independence and renaming/filtering

As we have seen in section 2, correctness of partial deduction requires that the atoms in $\mathscr{A}$ are independent. There are two ways to ensure the independence condition. The first one is to replace the atoms which are not independent by a more general atom (first proposed in Benkerimi & Lloyd (1990)). For example, replacing the dependent atoms $member(a, L)$ and $member(X, [b])$ by $member(X, L)$ in a set $\mathscr{A}$ removes the dependency; moreover the new set is closed with respect to all atoms in the old one. As discussed below, this approach can also be used to ensure global termination. However, it introduces precision loss as information about specific calls is disregarded; hence it can worsen the degree of global specialisation.

A better way to address the independence problem uses a so-called *renaming*

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

transformation, which renames every atom of $\mathscr{A}$ by giving it a *distinct* predicate symbol; the set of atoms to be specialised thus becomes independent *without* introducing any precision loss. For instance, given the dependent atoms $member(a, L)$ and $member(X, [b])$, renaming the second one into $member'(X, [b])$ removes the independence. The renaming transformation then also has to map the atoms inside the bodies of the residual program clauses of P' as well as atoms in queries for the specialised program to the correct versions. For example it should rename the query $\leftarrow member(a, [a, c]), member(b, [b])$ into $\leftarrow member(a, [a, c]), member'(b, [b])$.

Renaming can often be combined with so-called argument filtering to improve the efficiency of the specialised program. The basic idea is to filter out constants and functors and to keep only the variables as arguments. In terms of the fold/unfold transformation framework (Burstall & Darlington, 1977; Tamaki & Sato, 1984; Pettorossi & Proietti, 1994) it consists of defining new predicates and using it to fold occurrences in $\mathscr{A}$, P', and G. Considering the same examples, defining $mem_a(L) \leftarrow member(a, L)$ and $mem_b(X) \leftarrow member(X, [b])$, the dependent atoms $member(a, L)$ and $member(X, [b])$ are folded into the independent atoms $mem_a([a, c])$ and $mem_b(b)$, while the query is folded into $\leftarrow mem_a([a, c]), mem_b(b)$. Further details about filtering can be found in Gallagher & Bruynooghe (1991), Benkerimi & Hill (1993), Leuschel & Sørensen (1996) or Proietti & Pettorossi (1993). The specialisations shown in Safra & Shapiro (1986) strongly suggest that the authors already applied a form of argument filtering; it has also been referred to as "pushing down meta-arguments" in Sterling & Beer (1989) or "PDMA" in Owen (1989). In functional programming the term of "arity raising" has also been used. It has also been studied in an offline setting, where filtering is more complicated.

Renaming and filtering are used in a lot of practical approaches (e.g. Gallagher, 1991, 1993; Gallagher & Bruynooghe, 1991; Leuschel & De Schreye, 1995, 1998b; Leuschel *et al.*, 1998a) and adapted correctness results can be found in Benkerimi & Hill (1993). To avoid the need for a renaming transformation on queries to the specialised program, interface predicates are provided that define the original predicates in terms of the renamed ones.

5.2 Syntax-based Global Control

Having solved the independence problem without introducing any precision loss, we can now turn our attention to the problem of ensuring *closedness* and *global termination* while maximising the *degree of global specialisation.* In a so called monovariant analysis, the problem is solved by keeping at most one atom in $\mathscr{A}$ for each predicate. When several atoms occur with the same predicate symbol, they are replaced by a generalisation. This ensures that each predicate has at most one specialised version, ensuring correctness and – as there are no infinite chains of strictly more general expressions (Huet, 1980) – termination. However, as already said, generalising atoms introduces precision loss, hence it is worthwhile to consider *polyvariance*, the construction of several specialised versions of the same predicate. Deciding exactly how many versions is referred to as the *control of polyvariance problem.*

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

Let us examine how the closedness, global termination and the degree of global specialisation interact:

- *Closedness vs. Global Termination.*
  As we have seen in Procedure 1, closedness can be simply ensured by repeatedly adding the atoms which are not $\mathscr{A}$-closed to $\mathscr{A}$ and unfolding them. Unfortunately this process (first presented in Benkerimi & Hill (1993)) is not guaranteed to terminate.
- *Global Termination vs. Global Specialisation.*
  To ensure global termination one can use for the *revise* function in Procedure 1, a so-called *generalisation* operator, which generates a set of more general atoms. While replacing atoms by strictly more general ones introduces precision loss, it is sometimes essential to ensure termination.

The notion of generalisation can be formalised as follows:

Definition 13
Let $\mathscr{A}$ and $\mathscr{A}'$ be sets of atoms. Then $\mathscr{A}'$ is a *generalisation* of $\mathscr{A}$ iff every atom in $\mathscr{A}$ is an instance of an atom in $\mathscr{A}'$. A *generalisation operator* is an operator which maps every finite set of atoms to a generalisation of it which is also finite.

A generalisation operator is often referred to as an abstraction operator in the literature, but we think the term generalisation is more appropriate.

With $\mathscr{A}'$ a generalisation of $\mathscr{A}$, any set of clauses which is $\mathscr{A}$-closed is also $\mathscr{A}'$-closed. Using a generalisation operator as *revise* function in Procedure 1 does not guarantee global termination. But, if the procedure terminates then closedness is ensured, i.e. $P' \cup \{S\}$ is $\mathscr{A}$-closed (modulo renaming). With this observation we can reformulate the *control of polyvariance problem* as one of finding a *generalisation operator which maximises the global degree of specialisation while ensuring termination.* In the rest of this section we will survey methods that only consider the syntactic structure of the atoms to be specialised.

5.2.1 Most specific generalisation

Definition 14
The *most specific generalisation* or *least general generalisation* of a finite set of expressions E, denoted by $msg(E)$, is the most specific expression M such that all expressions in E are instances of M.

Some examples can be found in figure 6. The msg can be effectively computed (Lassez *et al.* (1988). The algorithm is also known as *anti-unification.* and dates back to Plotkin (1969) and Reynolds (1969). As already mentioned, given an expression A, there are no infinite chains of strictly more general expressions (Huet 1980).

This makes the msg well suited for use in a generalisation operator. One of the first generalisation operators was proposed in Benkerimi & Lloyd (1990). It applied the msg on atoms which have a common instance. As first pointed out in Martens *et al.* (1994), this does not ensure termination, as can be seen when specialising

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

A	B	$msg(\{A, B\})$
a	b	X
$p(a, b)$	$p(a, c)$	$p(a, X)$
$p(a, a)$	$p(c, c)$	$p(X, X)$
$p(0, s(0))$	$p(0, s(s(0)))$	$p(0, s(X))$
$q(0, f(0), 0)$	$q(a, f(a), f(a))$	$q(X, f(X), Y)$
$r(a)$	$r(s(a))$	$r(X)$

Fig. 6. Examples of *msg*.

Example 9 for the initial goal $\leftarrow rev(X, [], R)$ (no matter which terminating unfolding rule is used, all atoms in $\mathscr{A}'_i$ are independent, hence $generalise(\mathscr{A}'_i) = \mathscr{A}'_i$ and the set is growing forever).

A simple generalisation operator which ensures termination is obtained by imposing a finite maximum number of atoms in $\mathscr{A}_i$ for each predicate and using the msg to stick to that maximum (e.g. Martens *et al.*, 1994). However, the msg introduces precision loss and is applied at an arbitrary point. As illustrated in Martens *et al.* (1994, there will be cases where the msg is applied too early and precision loss is introduced that should have been avoided; in other cases, the msg is applied too late, resulting in too many uninteresting variants and code explosion.

5.2.2 Global Trees with wfos and wqos

We therefore need a more principled approach to global termination, much as we needed a more principled approach to local termination in section 4. Probably the first such solution, not depending on any *ad hoc* bound, is Martens & Gallagher (1995). The idea is to use the wfo approach also to ensure global termination. To do this, Martens & Gallagher (1995) proposed to structure the current atoms $\mathscr{A}_i$ (see Procedure 1) to be partially deduced as a so-called *global tree*, i.e. a tree whose nodes are labeled by atoms and where A is a child of B if specialisation of B leads to the specialisation of A, in the sense that $A \in leaves(unfold(P, B))$. This gives us a structure very similar to the SLDNF-trees encountered by the local control, and thus enables to apply wfo in much the same manner. In Leuschel *et al.* (1998a), this was extended to also accommodate wqos (and characteristic trees; which we discuss later).

Figure 7 contains a generic procedure based upon Martens & Gallagher (1995) and Leuschel *et al.* (1998a).

The procedure is parameterised by the unfold function $unfold(P, A)$, the predicate $covered(N, \gamma)$, the whistle function $whistle(N, \gamma)$ and the generalisation function $generalise(N, W, \gamma)$. The unfold function takes care of the local control and returns a finite SLDNF-tree. The predicate $covered(N, \gamma)$ decides whether there is already a partial deduction suitable for the atom $label(N)$. Termination and correctness require that it must return *true* when there is another marked node in the same branch

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

Procedure 2
Input: a program P and a set S of atoms of interest;
Output: A specialised program P' and a set of atoms $\mathscr{A}$;
let γ = a "global" tree consisting of a marked unlabeled root node R;
for each $A \in S$ **do**
    create in γ a new unmarked node C as a child of R;
    let $label(C) := A$
repeat
    pick an unmarked leaf node N in γ;
    if $covered(N, \gamma)$ **then** mark N as covered
    else
        let $W = whistle(N, \gamma)$;
        if $W \neq fail$ **then let** $label(N) := generalise(N, W, \gamma)$[4]
        else
            mark N as processed
            for all atoms $A \in leaves(unfold(P, label(N)))$ **do**
                create in γ a new unmarked node C as a child of N;
                let $label(C) := A$
until all nodes are marked;
let $\mathscr{A} := \{label(N) \mid N \in \gamma$ and N is not marked as covered$\}$;
let $P' := \bigcup_{A \in \mathscr{A}} resultants(unfold(P, A))$

Fig. 7. Generic tree-based partial deduction procedure.

labelled with a variant of $label(N)$ and that, whenever it returns *true*, the global tree γ has a marked node M such that $label(M)\theta = label(N)$ for some substitution θ. The whistle function $whistle(N, \gamma)$ prevents the growth of infinite branches in the global tree by using wfos or wqos; it raises an alarm by returning an ancestor node W of N in case N is not an admissible descendant of W (hence $label(W)$ has the same predicate symbol as $label(N)$) and *fail* otherwise. If N is not admissible, it has to be generalised. The generalisation function $generalise(N, W, \gamma)$ computes a generalisation of $label(N)$. To ensure termination, it must be a strict generalisation. Besides N it takes as parameters W and γ. The latter allows the function to return a generalisation that is admissible with respect to the whole branch ending in N. As the generalisation can now be covered by another marked node of the global tree, N should not yet be marked. If N is admissible, its label is unfolded and the leaves of the obtained SLDNF-tree are added as unmarked children of N while N is marked. Once all nodes are marked, the set $\mathscr{A}$ and the specialised program are extracted.

Observe that in the above procedure the generalisation operator of Definition 13 is split up into three components $covered(N, \gamma)$, $generalise(N, W, \gamma)$, and $whistle(N, \gamma)$. An instantiation of these three components that ensures correctness and terminations and uses the wqo $\trianglelefteq$ for $whistle(N, \gamma)$ is as follows (this is one of the possible settings in ECCE):

- $whistle(N, \gamma) = W$ iff W is the closest ancestor of L such that $label(W) \trianglelefteq label(L)$ and $label(L)$ is not strictly more general than $label(W)$,[5];
  $whistle(L, \gamma) = fail$ if there is no such ancestor.

[5] This latter test is required to avoid some technical difficulties with the way $\trianglelefteq$ treats variables; see Leuschel *et al.* (1998a) and Leuschel (1998a).

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

- $generalise(N, W, \gamma) = msg(label(N), label(W))$
- $covered(N, \gamma) = true$ if there is a node M in γ such that $label(M)$ is a variant of $label(N)$;
  $covered(N, \gamma) = false$ otherwise.

Discussion There are a few works within partial deduction of logic programs, in which the local and global control interact much more tightly, in the sense that the local control also takes information from the global control into account (Sahlin, 1993; Glueck *et al.*, 1996; De Schreye *et al.*, 1999; Vanhoof & Martens, 1997). Also observe that, in other programming paradigms such as supercompilation of functional languages (Turchin, 1986; Glueck & Sørensen, 1996; Sørensen *et al.*, 1996; Sørensen & Glück, 1999), historically there has not been a clear distinction between local and global control. In these settings (e.g. Sørensen & Glück, 1995; Sørensen *et al.*, 1996; Sørensen, 1998) there is only one big "global" tree which is then cut up into local trees during the code generation. This approach is also taken in the "compiling control" transformation of logic programs in Bruynooghe *et al.* (1989). In the future, it might be interesting to compare these two approaches systematically from a pragmatic point of view.

5.3 Computation-based global control

5.3.1 Characteristic trees

While the global trees of section 5.2.2 show the relationship between roots and leaves of constructed SLDNF-trees, the generalisation function which generalises the atoms is purely syntactical. It only takes into account the atoms as they appear in the global tree. However, the same two atoms can behave in a very similar way in the context of one program P_1, but in a very dissimilar fashion in the context of another program P_2. The syntactic structure of the two atoms being unaffected by the particular context, the generalisation function $generalise(N, W, \gamma)$ will thus perform exactly the same generalisation[6] within P_1 and P_2, even though very different action might be called for. A much more appealing approach, might therefore be to examine the SLDNF-trees generated for these atoms. These trees capture (to some depth) how the atoms behave computationally in the context of the respective programs. They also depict the specialisation that has been performed on these atoms. A generalisation operator which takes these trees into account will notice their similarity in the context of P_1 and their dissimilarity in P_2, and can therefore take appropriate actions in the form of different generalisations.

This observation lead to the definition of *characteristic trees*, initially presented in Gallagher & Bruynooghe (1991) and Gallagher (1991), and later exploited in Leuschel & De Schreye (1998a) and Leuschel *et al.* (1998a). In essence, characteristic trees abstract SLDNF-trees by only remembering, for the non-failing branches:

1. The position of the selected literals.

[6] Note, however, that $whistle(N, \gamma)$ can behave differently as γ will have a different structure.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

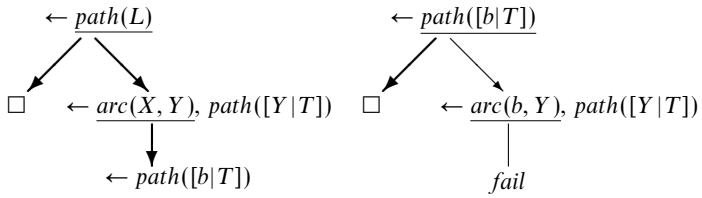

Fig. 8. SLD-trees for Example 12 .

2. An identification of the clauses C_1, C_2, ... used in the SLDNF-derivation of the branch.

We use $pos \circ cl$ to denote a derivation step that selects a literal at position pos and uses the clause identified by cl to compute a resolvent. A derivation or branch is represented as a sequence of derivation steps and a characteristic tree as a set of branches. The information in a characteristic tree is sufficient to rebuild the whole SLDNF-tree, hence it represents, directly or indirectly, all successful, failing and incomplete derivations. Two atoms with the same characteristic tree have so much in common (same number and "shape" of residual clauses) that one would expect that the same residual clauses can be used for both. We will discuss below whether and how that can be achieved. First we look at an example which shows that characteristic trees can also be useful for the whistle function $whistle(N, \gamma)$:

Example 12
Let P be the following definite program:

(1) $path([N]) \leftarrow$
(2) $path([X, Y|T]) \leftarrow arc(X, Y), path([Y|T])$
(3) $arc(a, b) \leftarrow$

Unfolding $\leftarrow path(L)$ (e.g., using an unfolding rule based on $\trianglelefteq$; see figure 8 for the SLD-trees constructed) will result in lifting $path([b|T])$ to the global level. Notice that we have a growth of syntactic structure ($path(L) \trianglelefteq path([b|T])$). However, one can see that further unfolding $path([b|T])$ results in an SLD-tree whose characteristic tree $\tau_B = \{\langle 1 \circ 1 \rangle\}$ is strictly smaller than the one for $path(L)$ (which is $\tau_A = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2, 1 \circ 3 \rangle\}$).

As the example illustrates the *growth* of syntactic structure can be accompanied by a *shrinking* of the associated SLDNF-trees. In such situations there is, despite the growth of syntactic structure, actually no danger of non-termination. A whistle function solely focussing on the syntactic structure would unnecessarily force generalisation, possibly resulting in a loss of precision. Other examples can be found in Leuschel *et al.* (1998a).

Incorporating characteristic trees into the global control has proven to be an elegant solution to avoid over-generalisation in several circumstances (when specialising meta-interpreters (Leuschel, 1997; VanHoof & Martens, 1997) or when specialising pattern matchers to obtain the "Knuth–Morris–Pratt" effect (Sørensen & Glück, 1999)).

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

A straightforward use of characteristic trees is as follows: classify atoms at the global control level by their associated characteristic tree and apply generalisation (*msg*) only on those atoms which have the same characteristic tree. This is basically the approach pursued in Gallagher & Bruynooghe (1991) and Gallagher (1991). Unfortunately, the approach has some problems. First, generalisation induces precision loss, even to the extent that the generalised atom has a different characteristic tree. Second, in case the number of distinct characteristic trees is not bounded, this approach will not terminate. We illustrate these two problems, and how to remedy them, in the next two subsections.

5.3.2 Preserving characteristic trees upon generalisation

Example 13
Let P be the program:

(1) $p(X) \leftarrow q(X)$
(2) $p(c) \leftarrow$

Let $\mathscr{A} = \{p(a), p(b)\}$. Assume that $q(X)$ is not unfolded. The atoms $p(a)$ and $p(b)$ have the same characteristic tree $\tau = \{\langle 1 \circ 1 \rangle\}$. Their msg, the atom $p(X)$ has a different characteristic tree, namely $\tau' = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2 \rangle\} \neq \tau$ and the specialisation for the atoms $p(a)$ and $p(b)$, due to the inapplicability of clause (2), is lost in the partial deduction of $p(X)$. More importantly, there exists *no* atom, more general than $p(a)$ and $p(b)$, which has τ as its characteristic tree.

The problem is that derivations that were absent in the original characteristic trees appear in the characteristic tree of the generalised atom. With negative literals, another source of difference is that a negative literal, ground (and selected) at some point in the original derivation is not necessarily ground, hence cannot be selected, in the SLDNF-tree of the generalised atom. More realistic examples can be found in Leuschel *et al.* (1998a) and Leuschel & De Schreye (1998a).

Two different solutions to this problem are:

1. *Ecological Partial Deduction.* (Leuschel, 1995; Leuschel *et al.*, 1998a)
   The basic idea is to use the characteristic tree as a recipe to build part of the SLDNF-tree (and to ignore the part not constructed by following the recipe). In Example 13, it means that the atom $p(X)$ is selected and clause (1) is used to construct a resolvent but that clause (2) is discarded as the branch using clause (2) is missing from the characteristic trees of $p(a)$ and $p(b)$. Extracting the residual clauses from the part of the SLDNF-tree that has been built yields the clause $p(X) \leftarrow q(X)$.

   The pruning possible for $p(a)$ and $p(b)$ is now preserved. However, the residual code is not correct for all instances of $p(X)$; it is only correct for those instances for which τ is a possible characteristic tree. Hence, in Algorithm 2, the function $covered(N, \gamma)$ should return *true* only if there is a node M such that $label(N)$ is an instance of $label(M)$ and if both have the same characteristic tree. In the example, the residual clause is correct for $p(a)$, $p(b)$, $p(d)$, but *neither* for $p(c)$ nor for $p(X)$. Note that this approach also works with

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

negative selected literals, and the above $covered(N, \gamma)$ test ensures that these negative literals do not become non-ground for the instances.

2. *Constrained Partial Deduction.* (Leuschel & De Schreye 1998a, Lafave & Gallagher, 1997)
   Whereas in standard partial deduction the members of $\mathscr{A}$ hence the roots of the SLDNF-trees are atoms, in constrained partial deduction, they are constrained atoms of the form $C \square A$, where A is an atom and C a constraint over some domain $\mathscr{D}$ (see Jaffar & Maher (1994) for details on constraint logic programming). Leuschel & De Schreye (1998a) use inequality constraints over the Herbrand universe. Considering again the generalisation of the characteristic trees for the atoms $p(a)$ and $p(b)$ of Example 13, they derive as generalisation the constrained atom $X \neq c \square p(X)$. This atom has the same characteristic tree as the original atoms. This also requires the $covered(N, \gamma)$ to be adapted, namely to check constraint entailment. However, constraints only appear during the partial deduction phase and the final specialised program is a pure logic program without constraints. Finally, this approach does not allow us to select negative literals, but is more powerful than the ecological partial deduction approach for definite programs, as the derived constraints are not just used locally to obtain the desired characteristic tree but they can be propagated globally to other atoms in $\mathscr{A}$ as well.

Recently, *trace terms* have also been used in place of characteristic trees (Gallagher & Lafave, 1996). Trace terms abstract away from the particular selection rule, making them more appealing in the context of pure logic programs. They also have the effect of providing a recipe during specialisation thus achieving the effect of ecological partial deduction, and they are easier to generate when using the cogen approach (Martin & Leuschel, 1999; Martin, 2000).

5.3.3 Ensuring termination without depth-bounds

It turns out that for a fairly *large class of realistic programs* (and unfolding rules), the characteristic tree based approaches described above only terminate when imposing a depth bound on characteristic trees. As the following simple example shows, this can lead to undesired results when the depth bound is actually required.

Example 14
A list type check on the second argument (the "accumulator") is added to the reverse program from Example 9

(1) $rev([], Acc, Acc) \leftarrow$
(2) $rev([H|T], Acc, Res) \leftarrow ls(Acc), rev(T, [H|Acc], Res)$
(3) $ls([]) \leftarrow$
(4) $ls([H|T]) \leftarrow ls(T)$

As can be noticed in figure 9, by using, e.g., determinate, $\trianglelefteq$-based, or well-founded unfolding we obtain an infinite number of different atoms, all with a different characteristic tree. Imposing a depth bound of say 100, we obtain termination;

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

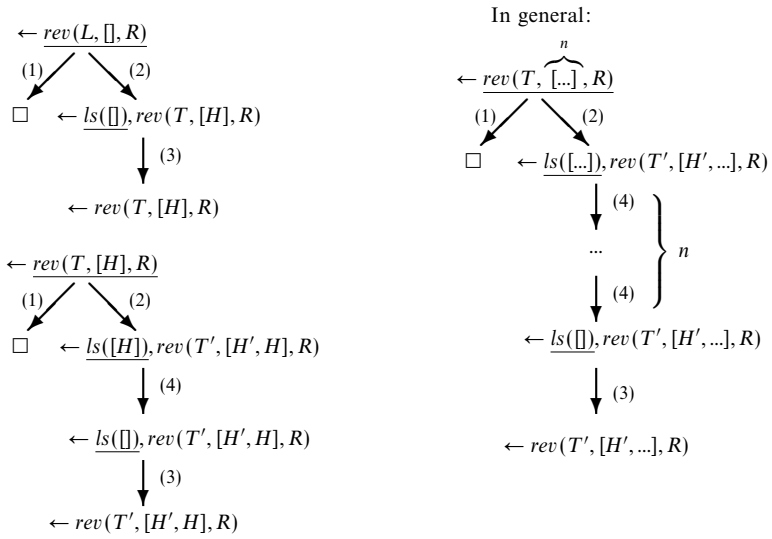

Fig. 9. SLD-trees for Example 14. .

however, 100 different characteristic trees (and instantiations of the accumulator) arise, and 100 different versions of *rev* are generated: one for each characteristic tree. The resulting specialised program is certainly far from optimal and clearly exhibits the ad hoc nature of the depth bound.

Situations like the above typically arise when some argument is growing with the level of recursion and when this argument has an influence on the characteristic tree of the SLDNF-tree built by the *unfold* function. With simple programs such as Example 9, the growing argument has no effect on the characteristic tree and it was believed for some time that the problem would not arise in "natural" logic programs. However, among larger and more sophisticated programs, cases like the above become more and more frequent, even in the absence of type-checking.

A solution to this problem is developed in Leuschel *et al.* (1998a), whose basic ingredients are as follows:

1. Register descendancy relationships among atoms and their associated characteristic trees at the global level, by putting them into a *global tree* (as in Section 5.2.2).
2. Watch over the evolution of the characteristic trees associated with atoms along the branches of the global tree to detect inadmissible branches (as in section 5.2.2). As suggested by figure 9, a measure is needed that can spot when a characteristic tree (piecemeal) "contains" characteristic trees appearing earlier in the same branch of the global tree. An extension of the homeomorphic embedding relation can be used for this (Leuschel *et al.*, 1998a). If such a situation arises – as it indeed does in Example 14 – one stops expanding the global tree, generalises the offending atoms, and produces a specialised procedure for the generalisation instead. Note that in this case, it

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

is actually impossible to preserve the characterstic trees upon generalisation, as the offending atoms will have different characteristic trees.

The techniques formally elaborated in Leuschel *et al.* (1998a) have led to the implementation of the ECCE system (Leuschel, 1996). The ECCE system also handles (declarative) Prolog built-ins; these are also registered within the characteristic trees (see Leuschel, 1997).

6 Conjunctive partial deduction and unfold/fold

6.1 Principles

Partial deduction, as defined above (i.e. based upon the Lloyd–Shepherdson framework (Lloyd and Shepherdson 1991)), specialises a *set of atoms*. Even though conjunctions of literals may appear within the SLDNF-trees constructed for these atoms, only atoms are allowed to appear at the global level. In other words, when we stop unfolding, every conjunction at the leaf is automatically split into its atomic constituents which are then specialised (and possibly further abstracted) separately at the global level. This restriction often considerably restricts the potential power of partial deduction, e.g. preventing the elimination of unnecessary variables (Proietti & Pettorossi, 1991b) (also called deforestation and tupling).

To overcome this limitation, Leuschel *et al.* (1996), Glück *et al.* (1996) and Leuchsel (1997) present a relatively small extension of partial deduction, called *conjunctive partial deduction*. This technique extends the standard partial deduction approach by considering sets $S = \{C_1, \ldots, C_n\}$ where the elements C_i are now *conjunctions* of atoms (to some extent negative literals can also be used within conjunctions) instead of just single atoms.

Now, as the SLDNF-trees constructed for each C_i are no longer restricted to having *atomic* top-level goals, resultants (cf. Definition 2) are not necessarily Horn clauses anymore: their left-hand side may contain a conjunction of literals. To transform such resultants back into standard clauses, conjunctive partial deduction requires a *renaming* transformation, from conjunctions to atoms, in a post-processing step. As with argument filtering, it can be formalised in the fold/unfold transformation framework by defining a new predicate and folding. The formal details are in Leuschel *et al.* (1996), Glück *et al.* (1996), Leuchsel (1997) and De Schreye *et al.* (1999). On the control side, there are two important issues that arise, which we address in the next two subsections.

6.2 Improved local specialisation

In addition to enabling tupling- and deforestation-like optimisations, conjunctive partial deduction also solves a problem already identified in Owen (1989). Take for example a metainterpreter containing the clause $solve(X) \leftarrow exp(X), clause(X, B), solve(B)$, where $exp(X)$ is an expensive test which for some reason cannot be (fully) unfolded. Here "classical" partial deduction faces an unsolvable dilemma, e.g. when specialising $solve(\bar{s})$, where $\bar{s}$ is some static input. Either it unfolds $clause(\bar{s}, B)$, thereby

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

propagating the static input $\bar{s}$ over to $solve(B)$, but at the cost of duplicating $exp(\bar{s})$ and most probably leading to inefficient programs (cf. Example 7). Or "classical" partial deduction can stop the unfolding, but then the partial input $\bar{s}$ can no longer be exploited inside $solve(B)$ as it will be specialised in isolation. Using conjunctive partial deduction however, we can be efficient *and* propagate information at the same time, simply by stopping unfolding and specialising the conjunction $C = clause(\bar{s}, B) \wedge solve(B)$. This will result in a specialised clause of the form: $solve(\bar{s}) \leftarrow exp(\bar{s}), conj\_cs(\bar{s})$, where $conj\_cs$ is the predicate defined by the clauses resulting from specialising the conjunction C. Experiments in Jørgensen *et al.* (1996) and Leuschel (1996) show that conjunctive partial deduction gives superior specialisation on programs as the above.

An additional benefit of this is that there is now much less need for non-determinate unfolding rules. For instance, while classical partial deduction with (almost) determinate unfolding performs badly on highly nondeterministic programs, this is no longer true for conjunctive partial deduction. The following table (extracted from Jørgensen *et al.*, 1996) for the "contains" benchmark underlines this:

System Type of PD Unfolding	ECCE Classical almost determinate	ECCE Classical non-determinate	MIXTUS Classical non-determinate	ECCE Conjunctive almost determinate
Speedup	1.18	11.11	6.25	9.09

6.3 Global control and implementation

Now, while it becomes easier to define an unfolding function that exploits all available information, there is a termination problem specific to conjunctive partial deduction. It lies in the possible appearance of ever growing conjunctions at the global level. To cope with this, generalisation in the context of conjunctive partial deduction must include the ability to *split* a conjunction into several parts, thus producing *subconjunctions* of the original one. A method to deal with this problem has been developed in Glück *et al.* (1996) and De Schreye *et al.* (1999), which treats the conjunction operator as an associative operator within $\trianglelefteq$ and then splits a conjunction according to the growth detected by $\trianglelefteq$ and computes the *msg* with the best matching subconjunction. This splitting reintroduces the problem that no information is exchanged between different components of a leaf, however, the components are now conjuncts instead of individual atoms.

For example, if the conjunction $C = p(X), q(f(X), s(0)), r(f(X)), s(X)$ has $C' = q(Z, 0), r(Z)$ as ancestor, then C' is embedded in C and one would split C into $C_1{=}p(X)$, $C_2{=}q(f(X), s(0)), r(f(X))$, $C_3{=}s(X)$. One would then compute the *msg* of C' and C_2, giving $C'' = q(Z, C), r(Z)$ as generalisation. Finally, as in classical partial deduction, one would then specialise C'' instead of C'.

Apart from the above modifications, the conventional control notions described earlier also apply in a conjunctive setting. Notably, the concept of characteristic trees can be generalised to handle conjunctions. The ECCE system (Leuschel, 1996),

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

discussed earlier, has been extended to handle conjunctive partial deduction and the extensive experiments conducted in Jørgensen *et al.* (1996) and Leuschel (1997) suggest that it was possible to consolidate partial deduction and unfold/fold program transformation, incorporating most of the power of the latter while keeping the automatic control and efficiency of the former.

6.4 Relationship to unfold/fold

Unfold/fold transformations of logic programs have been studied by Tamaki & Sato (1984) and Pettorossi & Proietti (1994), and were originally introduced by Burstall & Darlington (1977) in functional programming. The relation between unfold/fold and partial deduction has been a matter of research, discussion, and controversy over the years (Bossi *et al.*, 1990; Proietti & Pettorossi, 1993; Pettorossi & Proietti, 1994; Seki, 1993; De Schreye *et al.*, 1993). Within the fold/unfold transformation framework, there is work that aims at developing strategies that can be automated. For example, Pettorossi & Proietti (1994) describe a strategy for partial deduction. Their technique relies on a simple folding strategy involving no generalisation, so termination of the strategy is not guaranteed. Similar approaches are described in Proietti & Pettorossi (1991b, 1993) (in Proietti & Pettorossi (1993), generalisation is present in the notion of "minimal foldable upper portion" of an unfolding tree). Also, as unfold/fold transformations are equivalence preserving one needs a post-processing reachability analysis to delete dead code (for the queries under consideration). Such a reachability analysis is an integral part of partial deduction algorithms.

Another related approach is described in Boulanger & Bruynooghe (1993). The authors extend OLDT (Tamaki & Sato, 1986) to cope with conjunctions, similar to the way conjunctive partial deduction extends classical partial deduction. They then use abstract interpretation (in practice, generalisation is used as in partial deduction) to build a finite extended OLDT tree from which a specialised program is extracted. A major difference with (conjunctive) partial deduction is that a single global tree is built. The strategies needed to guide the construction of the optimal tree are lacking. It is plausible that the local and global control strategies developed for partial deduction could be translated into adequate strategies for building the extended OLDT tree.

In general, unfold/fold (together with a post-processing reachability analysis) can be seen to subsume both partial deduction and conjunctive partial deduction. However, from a practical point of view, partial deduction has advantages. Due to its more limited applicability, and its resulting lower complexity, the transformation can be more effectively and easily controlled. In fact, to our knowledge, no fully automatic unfold/fold systems are available for experimentation. However, some explicit strategies for unfold/fold transformation have been proposed and recently a semi-automatic system has been developed (Renault *et al.*, 1998). Let us consider some of the most well-known strategies: Loop Absorption and Generalisation (LAG) (Proietti & Pettorossi, 1993) and unfold-definition-fold (UDF) (Proietti and Pettorossi 1991b) (see also Pettorossi & Proietti (1994)). Both LAG and UDF use a class of computation rules, called *synchronised descent rules*; a heuristic tuned

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

towards foldability (and therefore, indirectly, termination of the strategy) and the generation of optimal transformed programs. However, neither LAG nor UDF guarantee termination in general. Instead, classes of programs are identified for which termination is ensured. As we have seen in this article, in partial deduction, methods have been proved to secure termination for *all* programs. Moreover, notions capturing the specialisation behaviour, such as characteristic trees, have been shown instrumental in providing precise generalisation. This level of technical detail has facilitated implementation, experimental evaluation and further improvements.

6.5 Relationship to other approaches

Techniques in Functional Programming. Partial deduction and related techniques in functional programming are often very similar (Glück & Sørensen (1994) (and cross-fertilisation has taken place). Actually, conjunctive partial deduction has in part been inspired by supercompilation of functional programming (Turchin, 1986; Glück & Sørensen, 1996; Sørensen *et al.*, 1996; Sørensen & Glück, 1999) (and by unfold/fold transformation techniques) and the techniques have a lot in common. However, there are still some subtle differences. Notably, while conjunctive partial deduction can perform deforestation *and* tupling, supercompilation is incapable of achieving tupling. On the other hand, the techniques developed for tupling of functional programs (Chin, 1993; Chin & Khoo, 1993) are incapable of performing deforestation.

The reason for this extra power conferred by conjunctive partial deduction, is that conjunctions with shared variables can be used both to elegantly represent *nested function calls*

$$f(g(X)) \quad \mapsto \quad g(X, \underline{ResG}),\ f(\underline{ResG}, Res)$$

as well as *tuples*

$$\langle f(X), g(X)\rangle \quad \mapsto \quad g(\underline{X}, ResG),\ f(\underline{X}, ResF)$$

or any mixture thereof. The former enables deforestation while the latter is vital for tupling, explaining why conjunctive partial deduction can achieve both.

Let us, however, also note that actually achieving the tupling or deforestation in a logic programming context can be harder. For instance, in functional programming we know that for the same function call we always get the same, unique output. This is often important to achieve tupling, as it allows one to replace multiple function calls by a single call. For example we can safely transform $fib(N) + fib(N)$ into `let` $X = fib(N)$ `in` $X + X$. However, in the context of logic programming, it is unsafe to transform the corresponding conjunction $fib(N, R1) \wedge fib(N, R2) \wedge Res\ is\ R1 + R2$ into $fib(N, R) \wedge Res\ is\ R + R$ unless it is proven or declared by the user that the relation $fib/2$ is functional in its first argument. Tupling in logic programming thus often requires one to establish *functionality* of the involved predicates. This can for instance be done via abstract interpretation (c.f. section 7) or via user declarations that are assumed to be correct or verified through analysis.

Furthermore, in functional programming, function calls cannot *fail* while predicate calls in logic programming can. This means that *reordering* calls in logic programming can induce a change in the termination behaviour; something which

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

is not a problem in (pure) strict functional programming. Unfortunately, reordering is often required to achieve deforestation or tupling. This means that to actually achieve deforestation or tupling in logic programming one often needs an additional analysis to ensure that termination is preserved (Bossi *et al*., 1995; Bossi & Cocco, 1996).

Partial evaluation of functional logic programs. Functional logic programming extends both logic and functional programming (Hanus, 1994). A lot of work has recently been carried out on partial deduction of such languages (Alpuente *et al*., 1996, 1997, 1998; Albert *et al*., 1998, 1999) (treating languages based on narrowing) and Lafave & Gallagher (1997) (treating languages based on rewriting). The developed control techniques have been strongly influenced by those developed for supercompilation of functional programs and (conjunctive) partial deduction of logic programs.

Compiling control. Another transformation technique close to both partial deduction and supercompilation is compiling control (Bruynoogheeeeee *et al*., 1989). A major difference with partial deduction is that the purpose is not to specialise a program based on the available static input but based on a better computation rule that reorders the execution of (generate and test) programs by performing tests as soon as their necessary inputs are available. To do so, the program is executed using a symbolic input (in fact, using an abstraction that abstracts ground terms by a "ground" symbol and leaves non-ground terms intact) and builds an initial segment of an infinite SLD-tree using an *oracle* to define the optimal execution order. The oracle either selects an atom for one unfolding step or for complete execution. In the latter case, the answers of the execution are abstracted using the ground symbol for ground terms (a more sophisticated abstraction, performing some generalisation on non-ground terms is needed in cases where this abstraction does not lead to a finite number of answers). The obtained incomplete tree is similar to the SLDNF-tree of partial deduction in that its nodes are goal statements. A difference with major partial deduction approaches is that a single global tree is built. Next, classes of similar nodes are identified in the tree. The similarity criterion is based on the selected atom and on the predicate symbols of the atoms presented in the nodes. Finally, the specialised program is extracted. In the context of partial deduction, that extraction can best be understood, as performing a local unfolding for each class (again using the oracle to guide the selection of atoms) until a leaf is reached that is a member of some class. At which point the resultants can be extracted and give rise to the specialised program. It is noteworthy that examples are treated which go beyond conjunctive partial deduction in the sense that goals, conjoined in a new predicate, can have – for some predicate symbols – a varying number of atoms. The atoms in question are joined in a list structure.

7 Discussion and conclusion

Research challenges

Despite over 10 years of research on logic program specialisation, there are still plenty of research challenges related to improving the actual specialisation capabilities.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

Below, we present what we believe to be the major research challenges for the coming years.

Control: Low-level cost model. Existing systems do not use a sufficiently precise model of the compiler of the target system to guide their decisions during specialisation. We have seen that determinate unfolding will usually prevent drastic slowdowns, but it is unable to exclude all slowdowns. Moreover, it is sometimes too conservative and prevents important improvements. While there is some recent work (Debray, 1997) to address this, it is a largely ignored area and some of the problematic issues raised in Venken & Demoen (1988) are still valid today.

A suitable *low-level cost model* would allow a partial deduction system to make more informed choices about the local control (e.g. is this unfolding step going to be detrimental to performance) and global control (e.g. does this extra polyvariance really pay off). However, such a low-level cost model will depend on both the particular Prolog compiler and on the target architecture and it is hence unlikely that one can find an appropriate mathematical theory. This means that further progress on the control of partial deduction will probably not come from ever more refined mathematical techniques such as new wqos, but probably more from heuristics and *artificial intelligence* techniques such as case-based reasoning or machine learning. For example, one might imagine a *self-tuning* system, which derives its own cost model of the particular compiler and architecture by trial and error. Such an approach has already proven to be highly successful in the context of optimising scientific linear algebra software (Whaley *et al.*, 2001). Some promising initial work on cost models for logic and functional programming has already been made in Albert *et al.* (2001) and Albert & Vidal (2001).

Predictable specialisation. Another drawback of existing specialisation systems (especially for online systems) is the lack of predictability for both the specialisation time and for the size of the generated residual program.

Indeed, while existing online systems and methods guarantee termination, their use sometimes results in code explosion without achieving substantial specialisation. One situation where this tends to happen is when the program to be specialised has a combination of arguments that can grow and shrink and when the initial atom to be specialised has partially instantiated parameters. The problem is that techniques such as $\trianglelefteq$ have, even given a fixed initial atom, no upper bound on the length of admissible sequences. For example, $\langle p(a,b), p(f(b), g(f(b), f(a)))\rangle$ is admissible wrt $\trianglelefteq$, as the growth of the second argument has been countered by the first argument (where we have $a \not\trianglelefteq f(b)$). A good example where such a behaviour can appear during specialisation is the "groundunify" benchmark within the DPPD library (Leuschel 1996), where two arguments are the terms to be unified (which are decomposed and thus usually shrink during specialisation) and another argument is the unifier so far (which will usually grow during specialisation). Using determinate unfolding for local control and $\trianglelefteq$ and characteristic trees for global control will lead to a global tree with 480 nodes and 85 specialised predicate definitions for

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

this benchmark. The specialisation effort here is out of proportion with the actual speedup obtained.

Developing control techniques with predictable and reasonable specialisation complexity is thus a worthwhile, but also challenging research objective. Alternatively, developing an *incremental partial deduction* approach could overcome these problems in some cases. Indeed, one could start by a very conservative partial deduction and then incrementally adapt the partial deduction, concentrating the efforts on the parts where improvements in efficiency or precision will arise. This could go hand-in-hand with a self-tuning system and a low-level cost model. Finally, as a side-benefit a user could stop the partial deduction at any point and still obtain a correct specialised program.

Improved precision: Combining program specialisation and abstract interpretation. As we have seen, $\trianglelefteq$ and characteristic trees provide a quite refined way to decide when the generalisation has to be applied. However, once a growth has been detected by $\trianglelefteq$, all of these existing specialisation techniques still rely on rather crude generalisation functions, such as *msg*, because the resulting generalisation has to be expressed as an atom, which implicitly represents all its instances. For instance, if we add the atom $A_2 = p(f(a))$ as a child of $A_1 = p(a)$ in a global tree then the homeomorphic embedding $\trianglelefteq$ will signal danger ($A_1 \trianglelefteq A_2$) and one can even pinpoint the extra $f(.)$ in A_2 as the potential source of non-termination. But the *msg* of A_1 and A_2 – the most specific expression which is more general than both A_1 and A_2 – is just $p(X)$ and no use of the information provided by $\trianglelefteq$ was made (nor is it possible to do so in classical partial deduction). In particular, atoms like $p(b)$ and $p(g(a))$ are also instances of $p(X)$, possibly leading to unacceptable losses of precision. In some cases the characteristic tree based global control will avoid these imprecisions. However, the present generalisation operation on the characteristic trees themselves is still a bit crude (common initial subsection). We think this problem in particular and other precision problems in general can be overcome by providing a better integration of partial deduction with abstract interpretation. This will also add other benefits, such as bottom-up success information propagation and success information propagation between atoms at the global level as well.

A full integration of partial deduction with *abstract interpretation* is thus another of the big challenges. Indeed, it is often felt that there is a close relationship between abstract interpretation and program specialisation. Some techniques preceding the recent advancements of partial deduction, notably compiling control (Bruynooghe *et al.*, 1989) and the work in Boulanger & Bruynooghe (1993) combine features of abstract interpretation with features of partial deduction. Recently, there has been a lot of interest in the integration of these two techniques (Jones, 1994, Leuschel & De Schreye, 1996, Puebla & Hermenegildo, 1996, Jones, 1997, Puebla *et al.*, 1997, Leuschel, 1998c, Gallagher & Peralta, 2001). The use of more refined abstract domains, improved bottom-up and side-ways information propagation, will improve specialisation and precision and opens up new areas for practical applications, such as *infinite model checking* (Leuschel & Massart, 1999, Leuschel and Lehmann, 2000b, Fioravanti *et al.*, 2001). In fact, such a combined approach

enables optimisations (and analysis) which cannot be achieved by either method alone (Leuschel and De Schreye 1996). Finally, having more precise generalisation capabilities might actually make the global and local control of partial deduction simpler, as much less precision would be lost if the control makes a "wrong" decision.

Tabling and constraints. Finally, features such as co-routining, constraints, and tabling provided by the latest generation Prolog systems, apart from being very useful in practice, also mean that declarative programming is now much more of a reality than in a classical Prolog environment. It is thus important that partial deduction be adapted to treat these features.

First, logic programming with inequality constraints provides a more sophisticated way to handle negated literals: by using so called *constructive negation* one can even specialise non-ground negative literals (Chan & Wallace, 1989). This idea was successfully used within the SAGE system (Gurr 1994a).

On the side of specialising arbitrary constraint logic programs themselves, we can mention the works of Smith & Hickey (1990), Smith (1991), Marriott & Stuckey (1993), Etalle & Gabbrielli (1996) and Bensaou & Guessarian (1998). Future work should advance the state of the art of specialising constraint logic programming to that for standard logic programming. First steps in that direction have been presented in Fioravanti *et al.* (1999, 2000).

In the context of tabled-evaluation of logic programs (Chen & Warren, 1996), some specialisation techniques have been successfully built into the execution mechanism itself (Dawson *et al.*, 1995), but there has been relatively little work on transforming or specialising tabled logic programs. Somewhat surprisingly, as shown in Leuschel *et al.* (1998b) and Sagonas & Leuschel (1998), tabled logic programming generates some new challenges to program transformation in general and partial deduction in particular. For example, contrary to the untabled setting, unfolding can transform a program terminating under tabled-evaluation into program that is non-terminating under tabled-evaluation.

Practical challenges: on the uptake of partial deduction

Despite some success stories and the increasing integration of partial deduction methods into compilers (e.g., the Mercury compiler specialises higher-order predicates such as map), the general uptake of partial deduction methods might be deemed disappointing. In the following we present some factors which we believe explain this situation:

- non-declarative features: most Prolog programs contain some form of non-declarative parts. Now, whereas systems such as MIXTUS or PADDY can handle such programs, non-declarative features impose severe restrictions on the specialiser, and the speedups obtained are often disappointing. In addition, most programs do not have a clear distinction between pure and impure parts, and it is thus difficult to apply systems such as SP or ECCE to large parts of the code.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press

To solve this problem, one might turn to more powerful, complementary analysis techniques, so as to lift some of the restrictions in the presence of impure features. E.g., one might integrate a partial evaluation system into Ciao Prolog where it could benefit from other analyses and/or optional user declarations. However, this is likely to involve considerable research and development effort.

Another solution is to promote a more declarative style of programming, more suitable for specialisation: e.g., programs written in Mercury, Gödel, or even pure Prolog with declarative built-ins and if-then-else and clearly separated i/o (or "declarative" i/o).

- For the offline approach, the lack of an implementation with a fully automatic bta, means that basically only expert users can use the current systems. However, as discussed earlier, some important steps towards automatisation of bta have recently been made and hopefully, they will soon become part of available systems.
- In principle, existing online systems such as MIXTUS and ECCE are fully automatic and can be used by a naïve user. However, as we have discussed above, for more involved programs, these systems can sometimes still lead to substantial code explosion and substantial specialisation times. Currently, to overcome this, user expertise is still required to fine tune the specialisation of the program at hand.
- Also, as we have seen above, existing systems do not use a sufficiently precise low-level cost model to guide the specialisation process. Consequently, they are unable to exclude anomalies such as slow-down of the specialised program.
- Finally, existing specialisers are not yet fully integrated within a programming environment. On the one hand, this means that it is more cumbersome to apply these tools (the user has to link up the specialised code with the rest of his application, the user has to know when parts of his application have to be respecialised,...). On the other hand, this means that currently specialisers are often only applied late in the development on already hand-optimised code. This makes the specialisers task more difficult and reduces the speedup and benefit.

  Thus, one of the practical challenges is to produce a partial deduction system that is fully integrated with a compiler, so that it can be easily used during and as part of the development process. Also, provide support for non-declarative parts and modules. Another difficulty is the interference with debugging, as users want to debug the code they wrote, not the specialised code.

However, we feel that it is possible to overcome the above obstacles and that in the not too distant future one could lift program specialisation towards more widespread practical use and realise its potential as a tool for systematic program development. As to the future of the off-line versus on-line debate, we believe that hybrid approaches might prove to be the way to go for many applications, delivering a good compromise between fast transformation speeds and precise specialisation. In fact, one approach which we have already found to be useful (Leuschel and

Lehmann 2000b) is to first perform an off-line specialisation followed by an on-line specialisation.

Acknowledgements

First, we would like to thank Danny De Schreye, André de Waal, Robert Glück, Bern Martens, and Morten Heine Sørensen for all the joint work and stimulating discussions. (Actually, some sentences from our joint works have probably made it into the paper.) We are grateful to Bart Demoen for his valuable feedback and insights, especially on Prolog performance issues. The authors would also like to thank the LOPSTR community for interest, enlightening comments, and discussions on many of the subjects covered in this paper. Finally, we would like to thank the anonymous referees for their very extensive and thorough feedback, which substantially helped us to improve paper.

References

Albert, E., Alpuente, M., Falaschi, M., Julián, P. and Vidal, G. (1998) Improving control in functional logic program specialization. In: G. Levi (ed.), *Static Analysis, SAS'98, Proceedings*, LNCS 1503, Springer-Verlag, pp. 262–277.

Albert, E., Alpuente, M., Hanus, M. and Vidal, G. (1999) A partial evaluation framework for curry programs. In: H. Ganzinger, D. A. McAllester and A. Voronkov (eds.), *Logic Programming and Automated Reasoning, LPAR'99, Proceedings*, LNCS 1705, Springer-Verlag, pp. 376–395.

Albert, E. and Vidal, G. (2001) Source-level abstract profiling for multi-paradigm declarative programs. In: A. Pettorossi (ed.), *Pre-Proceedings of 11th Int'l Workshop on Logic-based Program Synthesis and Transformation, LOPSTR'2001*.

Albert, E., Antoy, S. and Vidal, G. (2001) Measuring the effectiveness of partial evaluation in functional logic languages, *in* K-K. Lau (ed.), *Logic Based Program Synthesis and Transformation, LOPSTR 2000, Selected Papers*, LNCS 2042, Springer-Verlag, pp. 103–124.

Alpuente, M., Falaschi, M. and Vidal, G. (1996). Narrowing-driven partial evaluation of functional logic programs. In: H. Riis Nielson (ed.), *Programming Languages and Systems, ESOP'96, Proceedings*, LNCS 1058, Springer-Verlag, pp. 45–61.

Alpuente, M., Falaschi, M. and Vidal, G. (1998) Partial evaluation of functional logic programs. *ACM Trans. Program. Lang. Syst.* **20**(4), 768–844.

Alpuente, M., Falaschi, M., Julián, P. and Vidal, G. (1997) Spezialisation of lazy functional logic programs. *Proceedings of PEPM'97, ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, pp. 151–162.

Andersen, L. O. (1992) Partial evaluation of C and automatic compiler generation. In: U. Kastens and P. Pfahler (eds.), *Compiler Construction, CC'92, Proceedings*, LNCS 641, Springer-Verlag, pp. 251–257.

Andersen, L. O. (1994) *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen. (DIKU report 94/19).

Apt, K. R. (1990) Introduction to logic programming. In: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, North-Holland, pp. 495–574.

Apt, K. R. and Bol, R. N. (1994) Logic programming and negation: a survey. *J. Logic Program.* **19 & 20**, 9–72.

Aravindan, C. and Dung, P. M. (1994) Partial deduction of logic programs wrt well-founded semantics. *New Generat. Comput.* **13**: 45–74.

Beckman, L., Haraldson, A., Oskarsson, Ö. and Sandewall, E. (1976) A partial evaluator and its use as a programming tool. *Artif. Intell.* **7**, 319–357.

Benkerimi, K. and Hill, P. M. (1993) Supporting transformations for the partial evaluation of logic programs. *J. Logic & Computation*, **3**(5), 469–486.

Benkerimi, K. and Lloyd, J. W. (1990) A partial evaluation procedure for logic programs. In: S. Debray and M. Hermenegildo (eds.), *Proceedings North American Conference on Logic Programming*, MIT Press, pp. 343–358.

Bensaou, N. and Guessarian, I. (1998) Transforming constraining logic programs. *Theor. Comput. Sci.* **206**, 81–125.

Birkedal, L. and Welinder, M. (1994) Hand-writing program generator generators. In: M. Hermenegildo and J. Penjam (eds.), *Programming Language Implementation and Logic Programming, PLILP'91, Proceedings*, LNCS 844, Springer-Verlag, pp. 198–214.

Bol, R. (1993) Loop checking in partial deduction, *The Journal of Logic Programming* **16**(1&2), 25–46.

Bondorf, A. (1988) Towards a self-applicable partial evaluator for term rewriting systems. In: D. Bjørner, A. P. Ershov and N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, North-Holland, pp. 27–50.

Bondorf, A. (1989) A self-applicable partial evaluator for term rewriting systems. In: J. Diaz and F. Orejas (eds.), *TAPSOFT'89, Proceedings of the International Joint Conference on Theory and Practice of Software Development*, LNCS 352, Springer-Verlag, pp. 81–96.

Bossi, A. and Cocco, N. (1994) Preserving universal termination through unfold/fold. In: G. Levi and M. Rodriguez-Artalejo (eds.), *Algebraic and Logic Programming, ALP'94, Proceedings*, LNCS 850, Springer-Verlag, pp. 269–286.

Bossi, A. and Cocco, N. (1996) Replacement can preserve termination. In: J. Gallagher (ed.), *Logic Programming Synthesis and Transformation, LOPSTR'96, Proceedings*, LNCS 1207, Springer-Verlag, pp. 104–129.

Bossi, A., Cocco, N. and Dulli, S. (1990) A method for specialising logic programs. *ACM Trans. Program. Lang. Syst.* **12**(2), 253–302.

Bossi, A., Cocco, N. and Etalle, S. (1995) Transformation of left terminating programs: The reordering problem. In: M. Proietti (ed.), *Logic Program Synthesis and Transformation, LOPSTR'95, Proceedings*, LNCS 1048, Springer-Verlag, pp. 33–45.

Boulanger, D. and Bruynooghe, M. (1993) Deriving fold/unfold transformations of logic programs using extended OLDT-based abstract interpretation. *J. Symbolic Computation*, **15**(5&6), 495–521.

Bowers, A. F. and Gurr, C. A. (1995) Towards fast and declarative meta-programming. In: K. R. Apt and F. Turini (eds.), *Meta-logics and Logic Programming*, MIT Press, pp. 137–166.

Bruynooghe, M., De Schreye, D. and Krekels, B. (1989) Compiling control. *J. Logic Program.* **6**, 135–162.

Bruynooghe, M., De Schreye, D. and Martens, B. (1992) A general criterion for avoiding infinite unfolding during partial deduction. *New Generat. Computing*, **11**(1), 47–79.

Bruynooghe, M., Leuschel, M. and Sagonas, K. (1998) A polyvariant binding-time analysis for off-line partial deduction. In: C. Hankin (ed.), *Programming Languages and Systems, ESOP'98, Proceedings*, LNCS 1381, Springer-Verlag, pp. 27–41.

Bugliesi, M. and Russo, F. (1989) Partial evaluation in Prolog: Some improvements about cut. In: E. L. Lusk and R. A. Overbeek (eds.), *Logic Programming: Proceedings of the North American Conference*, MIT Press, pp. 645–660.

Burstall, R. M. and Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM*, **24**(1), 44–67.

Chan, D. and Wallace, M. (1989) A treatment of negation during partial evaluation. In: H. Abramson and M. Rogers (eds.), *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop*, MIT Press, pp. 299–318.

Chen, W. and Warren, D. S. (1996) Tabled evaluation with delaying for general logic programs. *J. ACM*, **43**(1), 20–74.

Chin, W.-N. (1993) Towards an automated tupling strategy. *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, pp. 119–132.

Chin, W.-N. and Khoo, S.-C. (1993) Tupling functions with multiple recursion parameters. in: P. Cousot, M. Falaschi, G. Filè, A. Rauzy (eds.), *Static Analysis, WSA'93, Proceedings*, LNCS 724, Springer-Verlag, pp. 124–140.

Codish, M. and Taboch, C. (1999) A semantic basis for the termination analysis of logic programs. *J. Logic Program.* **41**(1), 103–123.

Consel, C. and Danvy, O. (1993) Tutorial notes on partial evaluation. *Proceedings of ACM Symposium on Principles of Programming Languages* (*POPL'93*), ACM Press, pp. 493–501.

Dawson, S., Ramakrishnan, C. R., Ramakrishnan, I. V., Sagonas, K., Skiena, S., Swift, T. and Warren, D. S. (1995) Unification factoring for the efficient execution of logic programs. *Proceedings ACM Symposium on Principles of Programming Languages* (*POPL'95*), pp. 247–258.

Decorte, S., De Schreye, D. and Vandecasteele, H. (1999) Constraint-based termination analysis of logic programs. *ACM Trans. Program. Lang. Syst.* **21**(6), 1137–1195.

De Schreye, D. and Decorte, S. (1994) Termination of logic programs: The never ending story. *J. Logic Program.* **19 & 20**, 199–260.

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B. and Sørensen, M. H. (1999) Conjunctive partial deduction: Foundations, control, algorithms and experiments. *J. Logic Program.* **41**(2 & 3), 231–277.

de Waal, D. A. and Gallagher, J. (1994) The applicability of logic program analysis and transformation to theorem proving. In: A. Bundy (ed.), *Automated Deduction, CADE-12, Proceedings*, LNCS 814, Springer-Verlag, pp. 207–221.

Debray, S. (1997) Resource-bounded partial evaluation. *Proceedings PEPM'97, ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, pp. 179–192.

Denecker, M., Bruynooghe, M. and Marek, V. (2001) Logic programming revisited: logic programs as inductive definitions. *ACM Trans. Computational Logic*, **2**(4), 623–654.

Dershowitz, N. (1987) Termination of rewriting. *J. Symbolic Computation*, **3**, 69–116.

Dershowitz, N. and Jouannaud, J.-P. (1990) Rewrite systems. In: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Vol. B*, Elsevier/MIT Press, pp. 243–320.

Dershowitz, N. and Manna, Z. (1979) Proving termination with multiset orderings. *Comm. ACM*, **22**(8), 465–476.

Ershov, A. (1982) Mixed computation: Potential applications and problems for study. *Theor. Comput. Sci.* **18**, 41–67.

Etalle, S. and Gabbrielli, M. (1996) Transformations of CLP modules. *Theor. Comput. Sci.* **166**, 101–146.

Fioravanti, F., Pettorossi, A. and Proietti, M. (1999) Rules and strategies for contextual specialization of constraint logic programs. *Electr. Notes in Theor. Comput. Sci.* **30**(2).

Fioravanti, F., Pettorossi, A. and Proietti, M. (2000) Automated strategies for specializing

constraint logic programs. In: K-K. Lau (ed.), *Logic Based Program Synthesis and Transformation, LOPSTR 2000, Selected Papers*, LNCS 2042, Springer-Verlag, pp. 125–146.

Fioravanti, F., Pettorossi, A. and Proietti, M. (2001) Verifying CTL Properties of Infinite State Systems by Specializing Constraint Logic Programs. In: M. Leuschel, A. Podelski, C. R. Ramakrishnan and U. Ultes-Nitsche (eds.), *Proceedings Second International Workshop on Verification and Computational Logic, VCL'2001*, pp. 85-96.

Fujita, H. and Furukawa, K. (1988) A self-applicable partial evaluator and its use in incremental compilation. *New Generat. Comput.* **6**(2 & 3), 91–118.

Fuller, D. A. and Abramsky, S. (1988) Mixed computation of Prolog programs. *New Generat. Comput.* **6**(2 & 3), 119–141.

Fuller, D. A. F., Bocic, A. and Bertossi, L. E. (1996) Towards efficient partial evaluation in logic programming. *New Generat. Comput.* **14**, 237–259.

Gallagher, J. (1991) A system for specialising logic programs. *Technical Report TR-91-32*, University of Bristol.

Gallagher, J. (1993) Tutorial on specialisation of logic programs. *Proceedings PEPM'93, ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, pp. 88–98.

Gallagher, J. and Bruynooghe, M. (1991) The derivation of an algorithm for program specialisation. *New Generat. Comput.* **9**(3 & 4), 305–333.

Gallagher, J. and de Waal, D. A. (1992) Deletion of redundant unary type predicates from logic programs. In: K.-K. Lau and T. Clement (eds.), *Logic Program Synthesis and Transformation, Proceedings of LOPSTR'92*, Workshops in Computing, Springer-Verlag, pp. 151–167.

Gallagher, J. and Lafave, L. (1996) Regular approximations of computation paths in logic and functional languages. In: O. Danvy, R. Glück and P. Thiemann (eds.), *Partial Evaluation, International Seminar, Dagstuhl Castle, Selected Papers*, LNCS 1110, Springer-Verlag, pp. 115–136.

Gallagher, J. P. and Peralta, J. C. (2001) Regular tree languages as an abstract domain in program specialisation. *Higher Order and Symbolic Computation*, **14**(2–3), 143–172.

Glenstrup, A. J. and Jones, N. D. (1996) BTA algorithms to ensure termination of off-line partial evaluation. In: D. Bjørner, M. Broy and I. V. Pottosin (eds.), *Perspectives of System Informatics: Andrei Ershov Second International Memorial Conference, Proceedings*, LNCS 1181, Springer-Verlag, pp. 273–284.

Glück, R. and Sørensen, M. H. (1994) Partial deduction and driving are equivalent. In: M. Hermenegildo and J. Penjam (eds.), *Programming Language Implementation and Logic Programming, PLILP'94, Proceedings*, LNCS 844, Springer-Verlag, pp. 165–181.

Glück, R. and Sørensen, M. H. (1996) A roadmap to supercompilation. In: O. Danvy, R. Glück and P. Thiemann (eds.), *Partial Evaluation, International Seminar, Dagstuhl Castle, Selected Papers*, LNCS 1110, Springer-Verlag, pp. 137–160.

Glück, R., Jørgensen, J., Martens, B. and Sørensen, M. H. (1996). Controlling conjunctive partial deduction of definite logic programs. In: H. Kuchen and S. Swierstra (eds.), *Programming Languages: Implementations, Logics, and Programs, PLILP'96, Proceedings*, LNCS 1140, Springer-Verlag, pp. 152–166.

Gurr, C. A. (1994a) *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol.

Gurr, C. A. (1994b) Specialising the ground representation in the logic programming language Gödel. In: Y. Deville (ed.), *Logic Program Synthesis and Transformation, Proceedings of LOPSTR'93*, Workshops in Computing, Springer-Verlag, pp. 124–140.

Hanus, M. (1994) The integration of functions into logic programming. *J. Logic Programming*, **19 & 20**, 583–628.

Higman, G. (1952) Ordering by divisibility in abstract algebras. *Proc. Lond. Math. Soc.* **2**, 326–336.

Holst, C. K. (1989) Syntactic currying: yet another approach to partial evaluation. *Technical report*, DIKU, Department of Computer Science, University of Copenhagen.

Holst, C. K. and Launchbury, J. (1992) Handwriting cogen to avoid problems with static typing. Working paper.

Huet, G. (1980) Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, **27**(4), 797–821.

Jaffar, J. and Maher, M. J. (1994) Constraint logic programming: A survey. *J. Logic Program.* **19 & 20**, 503–581.

Jones, N. D. (1994) The essence of program transformation by partial evaluation and driving. In: N. D. Jones, M. Hagiya and M. Sato (eds.), *Logic, Language and Computation, Festschrift in Honor of Satoru Takasu*, LNCS 792, Springer-Verlag, pp. 206–224.

Jones, N. D. (1996) An introduction to partial evaluation. *ACM Comput. Surv.* **28**(3), 480–503.

Jones, N. D. (1997) Combining abstract interpretation and partial evaluation. In: P. Van Hentenryck (ed.), *Static Analysis, SAS'97, Proceedings*, LNCS 1302, Springer-Verlag, pp. 396–405.

Jones, N. D., Gomard, C. K. and Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation.* Prentice Hall.

Jones, N. D., Sestoft, P. and Søndergaard, H. (1989) Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP & Symbolic Computation*, **2**(1), 9–50.

Jørgensen, J. and Leuschel, M. (1996) Efficiently generating efficient generating extensions in Prolog. In: O. Danvy, R. Glück and P. Thiemann (eds.), *Partial Evaluation, International Seminar, Dagstuhl Castle, Selected Papers*, LNCS 1110, Springer-Verlag, pp. 238–262.

Jørgensen, J., Leuschel, M. and Martens, B. (1996) Conjunctive partial deduction in practice. In: J. Gallagher (ed.), *Logic Programming Synthesis and Transformation, LOPSTR'96, Proceedings*, LNCS 1207, Springer-Verlag, pp. 59–82.

Kleene, S. (1952) *Introduction to Metamathematics.* van Nostrand, Princeton, NJ.

Komorowski, J. (1982) Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog. *9th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 255–267.

Komorowski, J. (1992) An introduction to partial deduction. In: A. Pettorossi (ed.), *Meta-Programming in Logic, META-92, Proceedings*, LNCS 649, Springer-Verlag, pp. 49–69.

Kruskal, J. B. (1960) Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Trans. Am. Math. Soc.* **95**, 210–225.

Lafave, L. and Gallagher, J. (1997) Constraint-based partial evaluation of rewriting-based functional logic programs. In: N. Fuchs (ed.), *Logic Programming Synthesis and Transformation, LOPSTR'97, Proceedings*, LNCS 1463, pp. 168–188.

Lassez, J.-L., Maher, M. and Marriott, K. (1988) Unification revisited. In: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan-Kaufmann, pp. 587–625.

Leuschel, M. (1994) Partial evaluation of the "real thing". In: L. Fribourg and F. Turini (eds.), *Logic Programming Synthesis and Transformation – Meta-Programming in Logic, LOPSTR'94 and META'94, Proceedings*, LNCS 883, Springer-Verlag, pp. 122–137.

Leuschel, M. (1995) Ecological partial deduction: Preserving characteristic trees without constraints. In: M. Proietti (ed.), *Logic Program Synthesis and Transformation, LOPSTR'95, Proceedings*, LNCS 1048, Springer-Verlag, pp. 1–16.

Leuschel, M. (1996) The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~dtai`.

Leuschel, M. (1997) *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven. Accessible via `http://www.ecs.soton.ac.uk/~mal`.

Leuschel, M. (1998a) Improving homeomorphic embedding for online termination. In: P. Flener (ed.), *Logic-Based Program Synthesis and Transformation, LOPSTR'98, Proceedings*, LNCS 1559, Springer-Verlag, pp. 199–218.

Leuschel, M. (1998b) On the power of homeomorphic embedding for online termination. In: G. Levi (ed.), *Static Analysis, SAS'98, Proceedings*, LNCS 1503, Springer-Verlag, pp. 230–245.

Leuschel, M. (1998c) Program specialisation and abstract interpretation reconciled. In: J. Jaffar (ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming, JICSLP'98*, MIT Press, pp. 220–234.

Leuschel, M. (1999b) Logic program specialisation. In: J. Hatcliff, T. Æ. Mogensen and P. Thiemann (eds.), *Partial Evaluation: Practice and Theory, DIKU 1998 International Summer School*, LNCS 1706, Springer-Verlag, pp. 155–188 and 271–292.

Leuschel, M. and De Schreye, D. (1995) Towards creating specialised integrity checks through partial evaluation of meta-interpreters. *Proceedings PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, pp. 253–263.

Leuschel, M. and De Schreye, D. (1996) Logic program specialisation: How to be more specific. In: H. Kuchen and S. Swierstra (eds.), *Programming Languages: Implementations, Logics, and Programs, PLILP'96, Proceedings*, LNCS 1140, Springer-Verlag, pp. 137–151.

Leuschel, M. and De Schreye, D. (1998a) Constrained partial deduction and the preservation of characteristic trees. *New Generat. Comput.* **16**, 283–342.

Leuschel, M. and De Schreye, D. (1998b) Creating specialised integrity checks through partial evaluation of meta-interpreters. *J. Logic Program.* **36**(2), 149–193.

Leuschel, M. and Jørgensen, J. (1999) Efficient specialisation in prolog using the hand-written compiler generator Logen. *Electr. Notes in Theor. Comput. Sci.* 30(2).

Leuschel, M. and Lehmann, H. (2000a). Coverability of reset Petri nets and other well-structured transition systems by partial deduction, *in* J. Lloyd et al. (eds.), *Computational Logic, CL 2000, Proceedings*, LNAI 1861, Springer-Verlag, pp. 101–115.

Leuschel, M. and Lehmann, H. (2000b) Solving coverability problems of Petri nets by partial deduction. *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press, Montreal, Canada, pp. 268–279.

Leuschel, M. and Massart, T. (1999) Infinite state model checking by abstract interpretation and program specialisation. In: A. Bossi (ed.), *Logic-Based Program Synthesis and Transformation, LOPSTR'99, Selected Papers*. LNCS 1817, pp. 63–82.

Leuschel, M. and Sørensen, M. H. (1996) Redundant argument filtering of logic programs. In: J. Gallagher (ed.), *Logic Programming Synthesis and Transformation, LOPSTR'96, Proceedings*, LNCS 1207, Springer-Verlag, pp. 83–103. (Extended version as Technical Report CW 243, K.U. Leuven.)

Leuschel, M., De Schreye, D. and de Waal, A. (1996) A conceptual embedding of folding into partial deduction: Towards a maximal integration. In: M. Maher (ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, MIT Press, pp. 319–332.

Leuschel, M., Martens, B. and De Schreye, D. (1998a) Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Trans. Program. Lang. Syst.* **20**(1), 208–258.

Leuschel, M., Martens, B. and Sagonas, K. (1998b) Preserving termination of tabled logic programs while unfolding. In: N. Fuchs (ed.), *Logic Programming Synthesis and Transformation, LOPSTR'97, Proceedings*, LNCS 1463, pp. 189–205.

Levi, G. and Sardu, G. (1988). Partial evaluation of metaprograms in a multiple worlds logic language. *New Generat. Comput.* **6**(2 & 3), 227–247.

Lloyd, J. W. (1987) *Foundations of Logic Programming*. Springer-Verlag.

Lloyd, J. W. and Shepherdson, J. C. (1991) Partial evaluation in logic programming. *J. Logic Program.* **11**(3& 4), 217–242.

Lüttringhaus-Kappel, S. (1993) Control generation for logic programs. In: D. S. Warren (ed.) *Logic Programming, Proceedings of the Tenth International Conference on Logic Programming*, MIT Press, pp. 478–495.

Marchiori, E. and Teusink, F. (1995) Proving termination of logic programs with delay declarations. In: J. Lloyd (ed.), *Logic Programming, Proceedings 1995 International Symposium*, MIT Press, Cambridge, pp. 447–464.

Marriott, K. and Stuckey, P. (1993) The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, pp. 334–344.

Martens, B. (1994) *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K. U. Leuven.

Martens, B. and De Schreye, D. (1996) Automatic finite unfolding using well-founded measures. *J. Logic Program.* **28**(2):, 89–146.

Martens, B. and Gallagher, J. (1995) Ensuring global termination of partial deduction while allowing flexible polyvariance. In: L. Sterling (ed.), *Logic Programming, Proceedings 12th International Conference on Logic Programming*, MIT Press, pp. 597–613.

Martens, B., De Schreye, D. and Horváth, T. (1994) Sound and complete partial deduction with unfolding based on well-founded measures. *Theor. Comput. Sci.* **122**(1–2), 97–117.

Martin, J. (2000) *Judgement Day: Terminating Logic Programs*. PhD thesis, University of Southampton.

Martin, J. and King, A. (1997) Generating efficient, terminating logic programs. In: M. Bidoit and M. Dauchet (eds.), *Theory and Practice of Software Development, TAPSOFT'97, Proceedings*, LNCS 1214, Springer-Verlag, pp. 273–284.

Martin, J. and Leuschel, M. (1999) Sonic partial deduction. In: D. Bjørner, M. Broy and A. V. Zamulin (eds.), *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99, Proceedings*, LNCS 1755, Springer-Verlag, pp. 101–112.

Miniuissi, A. and Sherman, D. J. (1996) Squeezing intermediate construction in equational programs. In: O. Danvy, R. Glück and P. Thiemann (eds.), *Partial Evaluation, International Seminar, Dagstuhl Castle, Selected Papers*, LNCS 1110, Springer-Verlag, pp. 284–302.

Mogensen, T. and Bondorf, A. (1992) Logimix: A self-applicable partial evaluator for Prolog. In: K.-K. Lau and T. Clement (eds.), *Logic Program Synthesis and Transformation, Proceedings of LOPSTR'92*, Workshops in Computing, Springer-Verlag, pp. 214–227.

Mogensen, T. and Sestoft, P. (1997) Partial evaluation. In: A. Kent and J. G. Williams (eds.), *Encyclopedia of Computer Science and Technology*, Marcel Decker, New York, pp. 247–279.

Naish, L. (1993) Coroutining and the construction of terminating logic programs. *Australian Comput. Sci. Comm.* **15**(1), 181–190.

Neumann, G. (1990) Transforming interpreters into compilers by goal classification. In: M. Bruynooghe (ed.), *Proceedings 2nd Workshop on Meta Programming in Logic*, pp. 205–217.

Neumann, G. (1991) A simple transformation from Prolog-written metalevel interpreters into compilers and its implementation. In: A. Voronkov (ed.), *Logic Programming, First Russian Conference on Logic Programming – Second Russian Conference on Logic Programming, Proceedings*, LNCS 592, Springer-Verlag, pp. 349–360.

O'Keefe, R. (1985) On the treatment of cuts in Prolog source-level tools. *Proceedings 1985 Symposium on Logic Programming*, IEEE, pp. 68–72.

Owen, S. (1989) Issues in the partial evaluation of meta-interpreters. In: H. Abramson and M. Rogers (eds.), *Meta-Programming in Logic Programming, Workshop on Meta-Programming in Logic*, MIT Press, pp. 319–339.

Pettorossi, A. and Proietti, M. (1994) Transformation of logic programs: Foundations and techniques. *J. Logic Program.* **19& 20**, 261–320.

Plotkin, G. D. (1969) A note on inductive generalisation. In: B. Meltzer and D. Michie (eds.), *Machine Intelligence 5*, Edinburgh University Press, pp. 153–163.

Prestwich, S. (1992a) The PADDY partial deduction system. *Technical Report ECRC-92-6*, ECRC, Munich, Germany.

Prestwich, S. (1992b) An unfold rule for full Prolog. In: K.-K. Lau and T. Clement (eds.), *Logic Program Synthesis and Transformation, Proceedings LOPSTR'92*, Workshops in Computing, Springer-Verlag, pp. 199–213.

Prestwich, S. (1993) Online partial deduction of large programs. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93*, ACM Press, pp. 111–118.

Proietti, M. and Pettorossi, A. (1991) Semantics preserving transformation rules for Prolog. *Proceedings Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91*, *Sigplan Notices*, **26**(9), 274–284.

Proietti, M. and Pettorossi, A. (1991b) Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. In: J. Małuszyński and M. Wirsing (eds.), *Programming Language Implementation and Logic Programming, PLILP'91, Proceedings*, LNCS 528, Springer-Verlag, pp. 347–358.

Proietti, M. and Pettorossi, A. (1993) The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *J. Logic Program.* **16**(1 & 2), 123–162.

Przymusinska, H., Przymusinski, T. C. and Seki, H. (1994) Soundness and completeness of partial deductions for well-founded semantics. In: A. Voronkov (ed.), *Logic Programming and Automated Reasoning, LPAR'92, Proceedings*, LNCS 624, Springer-Verlag, pp. 1–12.

Puebla, G. and Hermenegildo, M. (1996) Abstract specialization and its application to program parallelization. In: J. Gallagher (ed.), *Logic Programming Synthesis and Transformation, LOPSTR'96, Proceedings*, LNCS 1207, pp. 169–186.

Puebla, G., Gallagher, J. and Hermenegildo, M. (1997) Towards integrating partial evaluation in a specialization framework based on generic abstract interpretation. In: M. Leuschel (ed.), *Proceedings ILPS'97 Workshop on Specialisation of Declarative Programs and its Application*, K.U. Leuven, Technical Report CW 255, pp. 29–38.

Renault, S., Pettorossi, A. and Proietti, M. (1998) *Design, Implementation, and Use of the MAP Transformation System.* Technical Report R. 491, IASI-CNR, Roma, Italy.

Reynolds, J. C. (1969) Transformational systems and the algebraic structure of atomic formulas. In: B. Meltzer and D. Michie (eds.), *Machine Intelligence 5*, Edinburgh University Press, pp. 135–151.

Romanenko, S. A. (1988) A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In: D. Bjørner, A. P. Ershov and N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, North-Holland, pp. 445–463.

Safra, S. and Shapiro, E. (1986) Meta interpreters for real. In: H.-J. Kugler (ed.), *Information Processing 86, Proceedings IFIP 10th World Computer Congress*, North-Holland, pp. 271–278.

Sagonas, K. and Leuschel, M. (1998) Extending partial deduction to tabled execution: Some

results and open issues. *ACM Comput. Surv.* **30** (Electronic Symposium on partial evaluation).

Sahlin, D. (1991) *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Swedish Institute of Computer Science.

Sahlin, D. (1993) Mixtus: An automatic partial evaluator for full Prolog. *New Generat. Comput.* **12**(1), 7–51.

Seki, H. (1993) Unfold/fold transformation of general programs for the well-founded semantics. *J. Logic Program.* **16**, 5–23.

Smith, D. A. (1991) Partial evaluation of pattern matching in constraint logic programming languages. *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, *Sigplan Notices*, **26**(9), 62–71.

Smith, D. A. and Hickey, T. (1990) Partial evaluation of a CLP language. In: S. Debray and M. Hermenegildo (eds.), *Logic Programming, Proceedings 1990 North American Conference*, MIT Press, pp. 119–138.

Somogyi, Z., Henderson, F. and Conway, T. (1996) The execution algorithm of Mercury: An efficient purely declarative logic programming language. *J. Logic Program.* **29**(1–3), 17–64.

Sørensen, M. H. (1998) Convergence of program transformers in the metric space of trees. In: J. Jeuring (ed.) *Mathematics of Program Construction, MPC'98, Proceedings*, LNCS 1422, Springer-Verlag, pp. 315–337.

Sørensen, M. H. and Glück, R. (1995) An algorithm of generalization in positive supercompilation. In: J. W. Lloyd (ed.), *Logic Programming, Proceedings of the 1995 International Symposium*, MIT Press, Portland, OR, pp. 465–479.

Sørensen, M. H. and Glück, R. (1999) Introduction to supercompilation. In: J. Hatcliff, T. Æ. Mogensen and P. Thiemann (eds.), *Partial Evaluation: Practice and Theory, DIKU 1998 International Summer School*, LNCS 1706, Springer-Verlag, pp. 246–270.

Sørensen, M. H., Glück, R. and Jones, N. D. (1996) A positive supercompiler. *J. Functional Program.* **6**(6), 811–838.

Sterling, L. and Beer, R. D. (1989) Metainterpreters for expert system construction. *J. Logic Program.* **6**(1 & 2), 163–178.

Takeuchi, A. and Furukawa, K. (1986) Partial evaluation of Prolog programs and its application to meta programming. In: H.-J. Kugler (ed.), *Information Processing 86, Proceedings IFIP 10th World Computer Congress*, North-Holland, pp. 415–420.

Tamaki, H. and Sato, T. (1984) Unfold/fold transformations of logic programs. In: S.-Å. Tärnlund (ed.), *Proceedings 2nd International Conference on Logic Programming*, pp. 127–138.

Tamaki, H. and Sato, T. (1986) OLD resolution with tabulation. In: E. Shapiro (ed.), *3rd International Conference on Logic Programming, Proceedings*, LNCS 225, Springer-Verlag, London, pp. 84–98.

Turchin, V. F. (1986) The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.* **8**(3), 292–325.

van Harmelen, F. (1989) The limitations of partial evaluation. In: P. Jackson and F. van Harmelen (eds.), *Logic-Based Knowledge Representation*, MIT Press, pp. 87–111.

Vanhoof, W. (2000) Binding-time analysis by constraint solving: a modular and higher-order approach for Mercury. In: M. Parigot and A. Voronkov (eds.), *Logic for Programming and Automated Reasoning, LPAR 2000 Proceedings*, LNAI 1955, Springer-Verlag, pp. 399–416.

Vanhoof, W. (2001) *Techniques for On- and Off-line Specialisation of Logic Programs*. PhD thesis, K. U. Leuven, Belgium.

Vanhoof, W. and Bruynooghe, M. (1999) Binding-time analysis for Mercury. In: D. De Schreye (ed.), *Logic Programming: The 1999 International Conference*, MIT Press, pp. 500–514.

Vanhoof, W. and Bruynooghe, M. (2001) Binding-time annotations without binding-time analysis. In: R. Nieuwenhuis and A. Voronkov (eds.), *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2001, Proceedings*, LNAI 2250, Springer-Verlag, pp. 707–722.

Vanhoof, W. and Martens, B. (1997) To parse or not to parse. In: N. Fuchs (ed.), *Logic Programming Synthesis and Transformation, LOPSTR'97*, LNCS 1463, Leuven, Belgium, pp. 322–342.

Venken, R. (1984) A Prolog meta interpreter for partial evaluation and its application to source to source transformation and query optimization. In: T. O'Shea (ed.), *Advances in Artificial Intelligence, Proceedings 6th European Conference on Artificial Intelligence, ECAI-84*, North-Holland, pp. 91–100.

Venken, R. and Demoen, B. (1988) A partial evaluation system for Prolog: Theoretical and practical considerations. *New Generat. Comput.* **6**(2 & 3), 279–290.

Whaley, R., Petitet, A. and Dongarra, J. (2001) Automated empirical optimizations of software and the atlas project. *Parallel Comput.* **27**(1–2), 3–35.

https://doi.org/10.1017/S147106840200145X Published online by Cambridge University Press