
**Ensuring Global Termination of Partial Deduction
while Allowing Flexible Polyvariance**

Bern Martens John Gallagher

December 1994

CSTR-94-016



University of Bristol

Department of Computer Science

Also issued as ACRC-94:CS-016

Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance

Bern Martens¹

John Gallagher

December 1994

Department of Computer Science
University of Bristol
Queen's Building
University Walk
Bristol BS8 1TR
U.K.

e-mail: {bern,john}@compsci.bristol.ac.uk

¹Supported during his stay at the University of Bristol by the European Community Human Capital and Mobility Network LPST. Permanent address: Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium, bern@cs.kuleuven.ac.be

Abstract

The control of polyvariance is a key issue in partial deduction of logic programs. Certainly, only finitely many specialised versions of any procedure should be generated, while, on the other hand, overly severe limitations should not be imposed. In this paper, well-founded orderings serve as a starting point for tackling this so-called “global termination” problem. Polyvariance is determined by the set of distinct “partially deduced” atoms generated during partial deduction. Avoiding ad-hoc techniques, we formulate a quite general framework where this set is represented as a tree structure. Associating weights with nodes, we define a well-founded order among such structures, thus obtaining a foundation for certified global termination of partial deduction. We include an algorithm template, concrete instances of which can be used in actual implementations, prove termination and correctness, and report on the results of some experiments. Finally, we conjecture that the proposed framework can support further advances towards (fully automatic) optimal program specialisation.

1 Introduction

As a major approach to program transformation and specialisation, partial evaluation has perhaps most intensively been studied in a functional programming context ([5], [15]). It was introduced to logic programming by [17] and has since then flourished also there (see e.g. [16], [10], [28]). Lately, it has become customary to speak about partial “deduction” rather than “evaluation” in a context of pure logic programs, and we will comply with that development in this paper, too.

A clear theoretical foundation for partial deduction was established in [23]. While clarifying a number of basic issues in the field, [23] did not address the question of *how* partial deduction should actually be performed. No actual procedure was included and no control issues were considered.

Providing adequate control for completely automatic partial deduction is however a crucial point in the development of a fully fledged, practical system for program specialisation. And one, so experience has shown, that is quite non-trivial to solve. Though there are obvious links between them, two different levels of consideration can be distinguished:

- A first concern pertains to the construction of SLD(NF)-trees that provide partial deductions for individual atoms. This is occasionally referred to as the *local* control level.
- The *global* level of control (on which this paper focuses), on the other hand, concerns itself with deciding for which atoms partial deductions have to be computed, making sure that the correctness conditions established in [23] are satisfied, and providing the right amount of polyvariance (i.e. generating different specialised procedures corresponding to a single general one in the original program).

In practice, these two levels are often intertwined, in the sense that decisions on one level influence those on the other. However, conceptually (and to a large extent also practically, in fact), they can be and occasionally have been separated to good effect.

At both levels, it is important to guide partial deduction in such a way that the efficiency gain resulting from its overall application is maximised. Several interesting heuristics and/or overall strategies, both for guiding unfolding and for steering polyvariance, have been proposed, often producing fine results on a range of examples, but no universally adequate methods have as yet emerged. (See e.g. [16] and [10] for two fairly recent accounts of these and related issues.)

At both levels also, it is crucial for a fully automatic method to proceed such that *termination* is guaranteed on all programs and queries without any user intervention. As far as tree construction is concerned, imposing a depth bound is an obvious way to certify its termination. However, this device has often been judged too crude, and research has investigated other methods, either to be combined with (some kind of) a depth bound (see e.g. [21], [34], [35], [2], [32], [3]) or not ([32], [3], [4], [25]).

Abstracting, then, from the local termination problem, in other words, assuming that for any atom and program, a partial deduction for that atom in that program can be delivered, the present paper sets out to investigate *global termination*. As indicated above, at the global level, the composition of the set of partially deduced atoms is handled. (As each such atom has an associated partial deduction, together finally constituting the whole specialised program, modulo perhaps some post-execution processing, this set also determines polyvariance.) Since

most procedures do not only add, but occasionally also remove elements from this set during partial deduction, there are two points to be addressed in global termination:

- The set should not be allowed to grow unboundedly.
- Loops that, while keeping the set bounded, repeatedly add and delete the same atom(s) should also be avoided.

An arbitrary bound on the set’s cardinality can apparently provide an answer to the first of these challenges. (The matter is complicated, though, by the fact that not just any set of atoms will be acceptable. See Theorem 2.5 below.) And, just as for local control, it is conceivable that a very large size bound combined with excellent optimality control (i.e. producing neither too little nor too much polyvariance) will perform very satisfactorily from a practical point of view. However, we feel that it is preferable to address global termination in a more principled way that better reflects the inherent logic of the problem, trying to remedy possible causes for non-termination instead of providing a symptomatic treatment. Doing so, we trust that the framework for termination to be presented below provides an interesting starting point for further refinements towards obtaining optimal polyvariance and maximally efficient specialised programs, within a system that is guaranteed to always terminate. The above mentioned papers [4] and [25] present an online approach to *local* termination that relies on concepts related to well-founded orderings. Our work on global termination in the present paper will draw on the same source of inspiration, but we will strive to make the ensuing presentation as much as possible self-contained.

The further layout of this paper, then, is as follows. Section 2 briefly recapitulates basic notions as well as a schematic algorithm for partial deduction of normal logic programs. In Section 3, we present our solution to the global termination problem, establishing a framework, including a generic algorithm and briefly discussing the performance of some of the latter’s concrete instances. Section 4 relates the approach to other work in partial deduction and partial evaluation, or even more general program transformation. Along the way, it attempts to assess the approach presented in this paper and suggests possible modifications and extensions, both to the framework and to concrete algorithms. Finally, we formulate a conclusion and assemble promising directions for further research in Section 5.

2 Partial Deduction

In what follows, we assume the reader to be familiar with the basic concepts in logic programming, as they are presented in e.g. [22]. Throughout, unless stated explicitly otherwise, the terms “(logic) program” and “goal” will refer to a *normal* logic program and goal, respectively.

First, we will designate an SLDNF-tree as *incomplete* when some of its leaves are neither success nor failure nodes, but an arbitrary goal where no literal has been selected for a further derivation step. Next, we reproduce the main definitions and an important theorem from [23].

Definition 2.1 Let P be a logic program, $G_0 = \leftarrow G$ a goal and G_0, G_1, \dots, G_n with $n > 0$, an SLDNF-derivation for $P \cup \{G_0\}$. Let $\theta_1, \dots, \theta_n$ be the corresponding sequence of substitutions and let G_n be $\leftarrow Q$.

$G\theta_1 \cdots \theta_n \leftarrow Q$ is called the *resultant* of the derivation G_0, G_1, \dots, G_n .

Definition 2.2 Let P be a program, A an atom and τ a finite (possibly incomplete) SLDNF-tree for $P \cup \{\leftarrow A\}$. Let $\{G_i | i = 1, \dots, r\}$ be the (non-root) leaves of the non-failing branches of τ and $\{R_i | i = 1, \dots, r\}$ the resultants corresponding to the derivations $\{\leftarrow A, \dots, G_i | i = 1, \dots, r\}$. The set $\{R_i | i = 1, \dots, r\}$ is called a *partial deduction of A in P* .

If $\mathcal{A} = \{A_1, \dots, A_s\}$ is a finite set of atoms, then a *partial deduction of \mathcal{A} in P* is the union of the partial deductions of A_1, \dots, A_s in P .

A *partial deduction of P wrt \mathcal{A}* is a logic program obtained from P by replacing the set of clauses in P whose head contains one of the predicate symbols appearing in \mathcal{A} (called the *partially deduced predicates*) by a partial deduction of \mathcal{A} in P .

Definition 2.3 Let \mathcal{A} be a finite set of atoms. We say \mathcal{A} is *independent* if no two atoms in \mathcal{A} have a common instance.

Definition 2.4 Let \mathcal{S} be a set of first-order formulas and \mathcal{A} a finite set of atoms. We say \mathcal{S} is *\mathcal{A} -closed* if each atom in \mathcal{S} containing a predicate symbol occurring in an atom in \mathcal{A} is an instance of an atom in \mathcal{A} .

Theorem 2.5 Let P be a logic program, G a goal, \mathcal{A} a finite, independent set of atoms, and P' a partial deduction of P wrt \mathcal{A} such that $P' \cup \{G\}$ is \mathcal{A} -closed. Then the following hold :

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

In other words, under the conditions stated in this theorem, computation with a partial deduction of a program is sound and complete with respect to computation with the original program. One of the tasks of the global control component of a partial deduction algorithm will be to ensure that the conditions imposed on \mathcal{A} are in fact verified. The elements of \mathcal{A} will be called *the partially deduced atoms*.

Let us now, starting from the above ingredients, set out to actually formulate a (very high level) algorithm for computing partial deductions. First, we introduce some auxiliary notions. The first of these abstracts from the details of local control, reflecting our intention not to address them in the present paper.

Definition 2.6 An *unfolding rule* U is a function which given a program P and an atom A , returns a finite set of resultants that is a partial deduction of A in P (*using U*). If \mathcal{A} is a finite set of atoms and P a program, then the set of resultants obtained by applying U to each atom in \mathcal{A} is called a *partial deduction of \mathcal{A} in P using U* .

We will use the notations $U(A, P)$ and $U(\mathcal{A}, P)$.

Definition 2.7 Let \mathcal{C} be a set of (normal) clauses or goals. Then we define $BA(\mathcal{C})$, the set of atoms (modulo variable renaming) appearing in the bodies of \mathcal{C} 's elements as:

$$BA(\mathcal{C}) = \{ A \mid \exists B \leftarrow Q, A, Q' \in \mathcal{C} \text{ or } \exists B \leftarrow Q, \sim A, Q' \in \mathcal{C} \} \text{ modulo variable renaming,}$$
where B may be absent and Q and Q' denote (possibly empty) conjunctions of literals.

Note that atoms being variants are considered identical. Of such variants, an arbitrary one is retained as an actual element of the resulting set. Likewise, comparisons between (sets of) atoms will, throughout this paper, be performed modulo variable renaming.

Algorithm 2.8

Input

a program P and goal G

Output

a set of partially deduced atoms \mathcal{A}

Initialisation

$\mathcal{A}_0 := \emptyset$

$\mathcal{A}_1 := BA(\{G\})$

$i := 1$

While $\mathcal{A}_{i-1} \neq \mathcal{A}_i$ **do**

$\mathcal{S} := \mathcal{A}_i \cup BA(U(\mathcal{A}_i, P))$ (*)

$\mathcal{A}_{i+1} := abstract(\mathcal{S})$ (**)

$i := i + 1$

Endwhile

$\mathcal{A} := \mathcal{A}_i$

Figure 1: A high level kernel algorithm for partial deduction.

The core of a high level algorithm for partial deduction is displayed in Figure 1. The algorithm calls for several comments. First, it explicitly seems to compute only a set of partially deduced atoms, rather than a partial deduction. But, obviously, such a set unambiguously determines its associated partial deduction in P (using U). Apart from partial deductions, other useful information may also be considered associated with the atoms in \mathcal{A} (see Section 4). Secondly, the algorithm incorporates only the backbone of a complete method for partial deduction (or program specialisation). Not included are e.g. static pre-execution renaming as in [2] and [26], dynamic renaming as in [1], and post-execution renaming and structure filtering as in [1], [12] or [13] (where it serves, among other purposes, to remove any remaining lack of independence from \mathcal{A}). Other post-execution processing might consist of reducing \mathcal{A} (and therefore polyvariance) for optimality reasons. Any of these can be added without invalidating what follows, although, in some cases, care should be taken not to introduce a new termination problem. Also not included is a prior analysis phase that derives supporting information for the main algorithm (e.g. through techniques related to abstract interpretation). We return to this point in Section 5. Next, as announced, we have not detailed how SLD(NF)-trees are to be constructed. Note that Definition 2.6 implicitly assumes that for every program and atom, a *finite* tree can be built. Apart from the techniques in (among others) the papers mentioned in Section 1, various versions of determinate unfolding, as discussed in [11] and [13], can provide inspiration. Finally, it is within the *abstract* operation that the essence of global control resides. In every iteration, taking the union in line (*) incorporates the striving towards closedness, as required by Theorem 2.5. Applying *abstract* in line (**) allows for refinements, possibly ensuring \mathcal{A}_i 's independence as in [2] and [26], imposing optimality control of polyvariance as in [13], and guaranteeing global termination. It is the latter aspect of *abstract* that we wish to concentrate on in this paper.

3 Ensuring Global Termination

3.1 Subset-wise founded marked trees

Two basic ingredients of our approach are *strict order relations*, denoted $>$, and *well-founded measures*. A strict order relation is an anti-reflexive, anti-symmetric and transitive binary relation. A (possibly partially) strictly ordered set, $V, >_V$, will be called an *s-poset*.

Definition 3.1 An s-poset $V, >$ is called *well-founded* if there is no infinite sequence of elements e_1, e_2, \dots in V such that $e_i > e_{i+1}$, for all $i \geq 1$.

Well-founded sets are a commonly used tool for proving termination of programs and production systems. A discussion and further references can be found in e.g. [7].

Definition 3.2 Let $V, >_V$ be an s-poset. A *well-founded measure*, f , on $V, >_V$ is a monotonic function from $V, >_V$ to some well-founded set $W, >_W$.

Below, we show how the \mathcal{A}_i sets in Algorithm 2.8 can be refined into finite tree-like structures and how proper manipulation of these structures guarantees termination of the resulting algorithm. Before reconsidering partial deduction in those terms, in the present subsection, we first introduce the structures we will use and establish some crucial properties of these. We set out with the following definition:

Definition 3.3 A *marked tree (m-tree)* is a (finitely branching) tree where nodes can be either *marked* or *unmarked*.

Using the standard terminology connected with trees, we speak about nodes and links, a root, branches, leaves, parent, child, ancestor and descendent nodes. Throughout the rest of this paper, we assume no order among the children of a node, and consider equality of m-trees modulo child permutations.

One may wonder what might be the relevance of the marks. When m-trees will be used to control partial deduction, a mark will basically indicate that the considered node contains an atom already partially deduced. See Section 3.2.

If μ is an m-tree, we denote the set of its nodes as N_μ , subset of some domain D_μ , the latter often left implicit. Let n_1 and n_2 be elements of N_μ , then we define $n_1 >_\mu n_2$ if and only if n_1 is an ancestor of n_2 in μ . Then $N_\mu, >_\mu$ is an s-poset. We can now introduce the following definition:

Definition 3.4 An m-tree μ is *subset-wise founded* if

1. There are finitely many disjoint sets, $C_1, \dots, C_N \subseteq D_\mu$, such that $N_\mu \subseteq \bigcup_{1 \leq i \leq N} C_i$.
2. For each $i = 1, \dots, N$, there exists a well-founded measure $f_i : C_i \cap N_\mu, >_\mu \rightarrow W_i, >_i$.

A pair of tuples $((C_1, \dots, C_N), (f_1, \dots, f_N))$, as in Definition 3.4 will be called a *class measure pair (cm-pair)*. We will occasionally speak about an m-tree being *subset-wise founded wrt a given cm-pair*. Note that any subtree of a subset-wise founded m-tree (wrt π) is also subset-wise founded (wrt π).

Example 3.5 Suppose that nodes in m-trees are labeled with atoms (as in Section 3.2 below) in a given language \mathcal{L} . Then a cm-pair can classify nodes according to their atom labels, for example by having one class per predicate symbol appearing in the label. Of course, more fine-grained distinctions are also possible, for a particular predicate symbol for instance further distinguishing between different top-level functors in their first argument. Possible measure functions can count (e.g.) functor occurrences in (some) arguments, thus mapping a node to a natural number reflecting the complexity of the atom in its label, or even impose lexical priorities among various, separately mapped, subsets of arguments. ($\mathbb{N}, >$, of course, is a well-founded set, and so is $\mathbb{N}^k, >_k$, where $>_k$ is the lexical ordering among k -tuples.) Various concrete objects similar to such cm-pairs are used to provide *local* control in [4], [25] and [26]. (In fact, below, we will slightly enhance measure functions of this type in order to cater for safe abstraction. See Example 3.21.)

Such an m-tree, then, is subset-wise founded with respect to such a cm-pair if argument structure decreases properly from one atom to the next *in the same class* along all branches of the tree.

Before we consider a concrete example, we formally define a particular class of measure functions as follows (taken from [4]):

Definition 3.6 Let $Term$ denote the set of terms in a language \mathcal{L} . We define the *functor norm* as the function $|\cdot| : Term \rightarrow \mathbb{N}$:

If $t = f(t_1, \dots, t_n), n > 0$
then $|t| = 1 + |t_1| + \dots + |t_n|$
else $|t| = 0$

The functor norm counts the number of functors (including “list” functors) in a given term. For example: $|[a, f(a), b]| = |[a, b, c, []]| = |[a, b, c, d|X]| = 4$.

Definition 3.7 Let p be a predicate of arity n and $S = \{a_1, \dots, a_m\}, 1 \leq a_k \leq n, 1 \leq k \leq m$, a set of argument positions for p . We define the *functor measure* with respect to p and S as the function

$|\cdot|_{p,S} : \{A | A \text{ is an atom with predicate symbol } p\} \rightarrow \mathbb{N}$:
 $|p(t_1, \dots, t_n)|_{p,S} = |t_{a_1}| + \dots + |t_{a_m}|$

Example 3.8 Let now \mathcal{L} be a language with two predicate symbols $p/2$ and $q/1$, providing labels for an m-tree as indicated above. Suppose that:

$C_1 = \{A | A \text{ is an atom in } \mathcal{L} \text{ with predicate symbol } p\}$
 $C_2 = \{A | A \text{ is an atom in } \mathcal{L} \text{ with predicate symbol } q\}$

and let $f_2 = |\cdot|_{q,\{1\}}$, then the tree in Figure 2 is subset-wise founded wrt $((C_1, C_2), (f_1, f_2))$ if $f_1 = |\cdot|_{p,\{2\}}$, but not if $f_1 = |\cdot|_{p,\{1\}}$ or $f_1 = |\cdot|_{p,\{1,2\}}$. However, the tree without node 6 is subset-wise founded wrt the latter cm-pair.

(Nodes in Figure 2 are identified through numbers; possible marks, being irrelevant for the considerations at hand, are not indicated. In the present context, the dotted box should be ignored.)

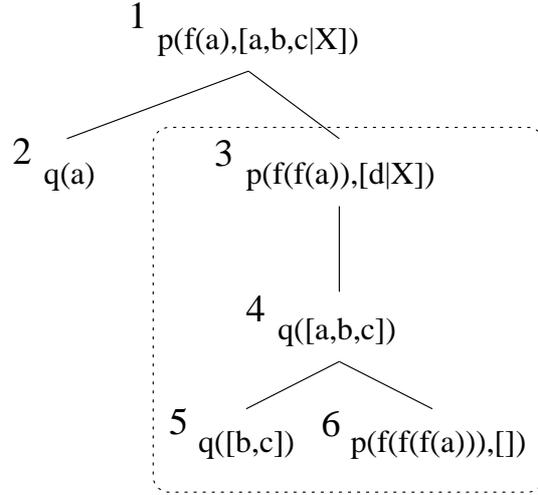


Figure 2: The m-tree referred to in Examples 3.8, 3.12 and 3.19.

Subset-wise foundedness characterises finiteness.

Proposition 3.9 An m-tree μ is finite iff it is subset-wise founded.

Proof First, if μ is finite, then $N_{\mu, >_{\mu}}$ is a well-founded set and the cm-pair in Definition 3.4 can be chosen with $N = 1$, $C_1 = N_{\mu}$ and f_1 the identity function on C_1 .

Conversely, suppose that μ is not finite. Then it follows from König's Lemma (see e.g. [8]) that there is at least one infinite branch in μ . This means that at least one $C_i \cap N_{\mu}$ contains an infinite sequence $n_{i_1} >_{\mu} n_{i_2} >_{\mu} \dots$. Applying f_i , we obtain an infinite descending sequence $f_i(n_{i_1}) >_i f_i(n_{i_2}) >_i \dots$ in W_i , contradicting the latter's well-foundedness. \square

In other words, when constructing m-trees, it suffices to ensure their subset-wise foundedness (with respect to a given cm-pair) to certify that they will be finite. Of course, in principle, this can also be attained with a single measure function used throughout the entire tree. But in the context of using m-trees to control partial deduction, the refined subset-wise approach is much more convenient and natural.

We can now consider the set of all m-trees that are subset-wise founded wrt a certain cm-pair, and turn it into a well-founded set by imposing a suitable partial order on it.

Definition 3.10 Let π be a cm-pair, then we denote as \mathcal{M}_{π} the set of all m-trees that are subset-wise founded wrt π .

Before we can move on, we need to introduce the following auxiliary concept.

Definition 3.11 If μ is a marked tree and n a node in μ , then we call the tree consisting of n together with all its descendants in μ and all the links connecting these nodes in μ , the terminal subtree of μ rooted in n .

Example 3.12 In Figure 2, the terminal subtree rooted in node 3 is contained within the dotted box.

Now, we can define:

Definition 3.13 Let π be a cm-pair and \mathcal{M}_π the set of m-trees, subset-wise founded wrt π . Then any function from \mathcal{M}_π to \mathcal{M}_π that performs (exactly) one of the following operations (on any μ in \mathcal{M}_π):

1. mark an unmarked node
2. add one or more child nodes to an unmarked leaf
3. remove a terminal subtree and replace its root $n \in C_i$ by an (unmarked) node $n' \in C_i$ such that $f_i(n) >_i f_i(n')$

is called a *direct tree transformer (dtt)*. We will denote the set of all direct tree transformers on a set \mathcal{M}_π as F_π .

In accordance with the possible operations listed in Definition 3.13, we will classify direct tree transformers as being of *type 1, 2 or 3*, respectively. Proposition 3.16 will show that the operations on the m-trees in the context of partial deduction can be understood in terms of dtts.

On \mathcal{M}_π , we finally define the desired partial order, reflecting dtt applications, as follows:

Definition 3.14 Let π be a cm-pair. We define $>_\pi$ on \mathcal{M}_π as the transitive closure of the following relation R :

$$R = \{(\mu_1, \mu_2) \in \mathcal{M}_\pi \times \mathcal{M}_\pi \mid \exists f \in F_\pi : \mu_2 = f(\mu_1)\}$$

The following crucial theorem can now be proved:

Theorem 3.15 Let π be a cm-pair. Then $\mathcal{M}_\pi, >_\pi$ is a well-founded set.

Proof First, observe that $>_\pi$ is a strict (partial) order on \mathcal{M}_π . Indeed, $>_\pi$ is transitive by definition. That it is also anti-reflexive and anti-symmetric follows from its well-foundedness, demonstrated below.

So, let us show that there can be no infinite descending sequence in $\mathcal{M}_\pi, >_\pi$. First, observe that $\mu >_\pi \mu'$ iff $\exists f_1 \dots f_n \in F_\pi : \mu' = f_n(\dots f_1(\mu) \dots)$. Slightly simplifying the notation, we will write $f_n \dots f_1(\mu)$. So, we have to show that there can be no infinite sequence $\mu, f_1(\mu), f_2 f_1(\mu), \dots$ in \mathcal{M}_π . In other words, we must show that the number of direct tree transformers that can successively be applied to any μ is finite, and, without loss of generality, we may assume that μ consists of a single unmarked node n .

Now, first, before a dtt of another type is applied, just a single type 1 dtt is applicable to μ . This provides the base case for a straightforward induction argument showing that after a finite number of dtt applications (say f_1 to f_l), the number of type 1 dtts consecutively applicable *before a dtt application of type 2 or type 3*, is finite. (This number is equal to the number of unmarked nodes in $f_l \dots f_1(\mu)$.) It remains to be shown that only a finite number of type 2 or type 3 dtts are applicable.

We first argue that after a finite number of dtt applications, only a finite number of type 2 dtts can consecutively be applied *before a dtt application of type 3*. Indeed, every type 2 dtt adds one or more nodes to the tree it is applied to. This can only happen finitely often since the elements of $\mathcal{M}_\pi, >_\pi$ are subset-wise founded m-trees. We are therefore left with the task of showing that, starting from μ , only a finite number of type 3 dtts can be applied.

The proof involves several arguments by induction. The main induction is on the number of classes in π . Both the proof of the base case and the proof of the induction step in that overall induction involve a subsidiary induction, in the latter case in its turn requiring a subordinate induction argument to show that its induction step holds.

First, then, suppose that $\pi = (C, f)$, i.e. it consists of a single class C and associated measure function $f : C \rightarrow W, >_W$. We want to prove that from μ (consisting of a single node $n \in C$), only a finite number of type 3 dtts (on \mathcal{M}_π) can be applied. The proof of this base case in the overall induction is through induction on the number of elements in the longest sequence $f(n) >_W w_1 >_W w_2 >_W \dots$ in W . Let us denote this number as $l(n)$. If $l(n) = 1$, only a single dtt (of type 1) is applicable and the result is immediate. Suppose now that $l(n) > 1$ and that the result holds for any n' such that $l(n) > l(n')$ (this is the induction hypothesis for the current subsidiary induction proof). Applying a type 3 dtt to μ immediately (or after first applying a type 1 dtt) reduces the problem to the induction hypothesis, since it replaces n by a node m such that $f(n) >_W f(m)$ (and therefore $l(n) > l(m)$). Alternatively, we can first apply a type 2 dtt to obtain a two-layer tree μ' with root n and leaves n_1, \dots, n_N such that $\forall 1 \leq i \leq N : f(n) >_W f(n_i)$. Now we can use the induction hypothesis on the child nodes, and derive that after a finite number of type 3 dtt applications (only affecting proper subtrees of the entire tree), the only remaining possibility involves removing the entire tree with root n (and replacing n by a node m such that $f(n) >_W f(m)$). However, as observed above, applying a type 3 dtt that does this, reduces the situation to one covered by the induction hypothesis, and we conclude that after it, only finitely many type 3 dtt applications can follow. This concludes the proof of the base case in the main induction argument.

Let us now assume that the result holds for any π with at most N classes and associated measure functions (this is the induction hypothesis of the main induction argument). We show that it is then also valid for all π with $N + 1$ classes and measure functions. Now, let $\pi = ((C_1, \dots, C_{N+1}), (f_1, \dots, f_{N+1}))$ and μ consist of node $n \in C_M$ ($1 \leq M \leq N + 1$). Again the proof proceeds through a subsidiary induction argument, this time on L_M , upper bound to $l_M(m)$ (where $l_M(m)$ is defined similarly to $l(n)$ above) for any node $m \in C_M$ that appears in the m-trees produced by consecutively applying a series of dtts s to an m-tree ν in \mathcal{M}_π . In other words, we will prove that for any value of this upper bound, if ν and s satisfy it, only finitely many type 3 dtts can occur in s . Since $l_M(n)$ provides such an upper bound for any s applied to μ , the desired result follows from this. As before, it is sufficient to consider a starting tree ν consisting of a single node n_ν .

The base case of this subsidiary induction is $L_M = 0$. This means that in none of the constructed trees a node in C_M appears. Therefore, all these trees are subset-wise founded with respect to $((C_1, \dots, C_{M-1}, C_{M+1}, \dots, C_{N+1}), (f_1, \dots, f_{M-1}, f_{M+1}, \dots, f_{N+1}))$. This cm-pair contains only N classes and measure functions. The result therefore follows from the induction hypothesis of the main induction argument.

Let us then consider $L_M = k$ and assume that for any starting tree ν' and series s' of dtts such that all nodes $m \in C_M$ in any produced tree have $l_M(m) < k$, s' is guaranteed to contain only finitely many type 3 dtts. (Let us call this induction hypothesis H .) Now either $n_\nu \in C_M$ or this is not the case. Suppose first it is. Then $l_M(n_\nu)$ provides an upper bound for the $l_M(m)$ values in all trees constructible from ν through any series s . Therefore, the only interesting case is $l_M(n_\nu) = k$. Leaving the trivial operation of marking n_ν out of consideration, a series s can start with either a type 2 or a type 3 dtt. In the latter case, the problem is reduced to the induction hypothesis H , since we must have $l_M(n_\nu) > l_M(r)$ where r is the new root

node, replacing n_ν . In the former case, finitely many child nodes are added to n_ν . Again apart from the possible marking of n_ν , any dtt occurring in s before a type 3 dtt removing n_ν as discussed above, can be considered as occurring in exactly one subseries of s starting from one of the children c of n_ν . However, since all produced trees must be subset-wise founded wrt π , any C_M node in any of the (sub)trees produced in any of these series must have an l_M value strictly smaller than k . The result now again follows from H .

Finally, then, suppose $n_\nu \notin C_M$. To deal with this case, we need a final induction argument, reasoning on the number of times a series s introduces a node $n_M \in C_M$ with $l_M(n_M) = k$ in a constructed m-tree. The base case of this final induction is when this never happens. But then we can use H to conclude that s can contain only finitely many type 3 dtts. Now suppose that any s which does this t times can contain only finitely many type 3 dtts. (Let this be induction hypothesis H' .) Then an s' indulging in this practice $t+1$ times can be considered as constituted of a subseries doing it at most t times, one type 2 dtt that introduces at least one such node, and a disjoint subseries operating from the $t+1$ st node. The first subseries contains only finitely many type 3 dtts due to H' , and the latter operates starting from a tree with root node $r \in C_M$ such that $l_M(r) = k$ and, above, we have shown that such a series is also bound to contain only finitely many type 3 dtts. \square

3.2 Global termination through subset-wise founded m-trees

Let us now return to the proper topic of this paper: (globally) controlling partial deduction. In Algorithm 2.8, termination of the While-loop and global control in general depend on the successive \mathcal{A}_i sets. We now propose to refine these sets into m-trees, labeling the latter's nodes with the atoms in the corresponding set. In each iteration, the given m-tree, subset-wise founded wrt a certain cm-pair π , will be transformed into a successor m-tree, likewise subset-wise founded wrt π and obtainable from its predecessor through a finite number of direct tree transformers. Therefore, it will be smaller in the associated $>_\pi$ order on \mathcal{M}_π and since that is a well-founded set, the algorithm will be bound to terminate.

The resulting algorithm, formulated in terms of formal concepts to be introduced below, is depicted in Figure 3 (page 15). The reader may find it helpful to already now take a first look at the informal description of the algorithm's operation that precedes Example 3.25.

The m-trees considered in this section will have nodes that are couples (A, i) where A is an atom in the language underlying the program and goal to be partially deduced, and i is a(n arbitrary) unique identifier, specific to this particular node in the tree. Let \mathcal{M} be the set of all such m-trees. If μ is an m-tree, then N_μ (as before) will denote the set of its nodes, L_μ the set of its leaves, L_μ^- the set of its non-marked leaves and \mathcal{A}_μ the set of atoms appearing in its nodes. Finally, for a node $n = (A, i)$, $atom(n)$ will denote A . We introduce the following *m-tree transforming operators*:

- *extend a leaf*

Let $\mu \in \mathcal{M}$, $l \in L_\mu^-$ and \mathcal{A} be a set of atoms, then $EL(\mu, l, \mathcal{A})$ is an m-tree μ' , the extension of μ obtained by marking l and adding a child node (A_i, i) of l for every atom $A_i \in \mathcal{A}$.

- *extend (several leaves in) a tree*

Let $\mu \in \mathcal{M}$, $l_1 \neq \dots \neq l_n \in L_\mu^-$ and $\mathcal{A}_1, \dots, \mathcal{A}_n$ be sets of atoms, then

$$ET(\mu, \{(l_1, \mathcal{A}_1), \dots, (l_n, \mathcal{A}_n)\}) = EL(\dots EL(EL(\mu, l_1, \mathcal{A}_1), l_2, \mathcal{A}_2) \dots, l_n, \mathcal{A}_n)$$

- *cut an unmarked leaf, if present*

Let $\mu \in \mathcal{M}$ and $l \notin N_\mu \setminus L_\mu^-$, then $CL(\mu, l)$ is:

- the m-tree obtained from μ by deleting l (and its incoming link) if $l \in L_\mu^-$
- μ if $l \notin N_\mu$

- *cut several leaves*

Let $\mu \in \mathcal{M}$ and $l_1, \dots, l_n \notin N_\mu \setminus L_\mu^-$, then

$$CLT(\mu, \{l_1, \dots, l_n\}) = CL(\dots CL(CL(\mu, l_1), l_2) \dots, l_n)$$

- *cut and replace a subtree rooted in a non-leaf node*

Let $\mu \in \mathcal{M}$ and A be an atom, then $CN(\mu, n, A)$ is:

- the m-tree obtained from μ by deleting μ 's terminal subtree rooted at n , and replacing n in μ by an (unmarked) leaf (A, i_A) if $n \in N_\mu$
- μ if $n \notin N_\mu$

- *cut and replace several subtrees*

Let $\mu \in \mathcal{M}$, $n_1, \dots, n_k \in N_\mu$ and A_1, \dots, A_k be atoms, then

$$CNT(\mu, \{(n_1, A_1), \dots, (n_k, A_k)\}) = CN(\dots CN(CN(\mu, n_1, A_1), n_2, A_2) \dots, n_k, A_k)$$

- *cut (and replace non-leaf) parts of a tree*

Let $\mu \in \mathcal{M}$, $L \subseteq L_\mu^-$, $n_1, \dots, n_k \in N_\mu$ and A_1, \dots, A_k be atoms then

$$CT(\mu, L, \{(n_1, A_1), \dots, (n_k, A_k)\}) = CLT(CNT(\mu, \{(n_1, A_1), \dots, (n_k, A_k)\}), L)$$

One can verify that the above operators are well-defined and that their result is determined uniquely modulo node identifiers.

Extending the m-tree, through an *ET* operation, will correspond to line (*) in Algorithm 2.8. Subsequently cutting (and replacing) parts of the result, using *CT*, will incorporate the *abstract* operation in line (**).

It is interesting to note that:

Proposition 3.16 Let $\pi = ((C_1, \dots, C_k), (f_1, \dots, f_k))$ be a cm-pair, $\mu, \mu' \in \mathcal{M}_\pi$ and μ' derivable from μ through the application of one or more of the above m-tree transforming operators. If upon every application of *CN*, the node n' that replaces the deleted terminal subtree root $n \in C_i$ is also in C_i and $f_i(n) >_i f_i(n')$, then μ' can be constructed from μ through the application of zero or more dtts on \mathcal{M}_π .

Proof Immediate from Definition 3.13 and the operators' definitions. □

Before we can present the actual algorithm, we need to define the following concepts:

Definition 3.17 Let $\pi = ((C_1, \dots, C_k), (f_1, \dots, f_k))$ be a cm-pair, μ an m-tree and $n, m \in N_\mu$ ($n \neq m$). We say that n π -covers m if the following two conditions are satisfied:

1. $n >_\mu m$
2. $\exists l \leq l \leq k : n, m \in C_l$

Definition 3.18 Let π be a cm-pair, μ an m-tree and $n, m \in N_\mu$. We say that n is the *closest π -covering ancestor of m in μ* , $cca_\pi(\mu, m)$, if the following two conditions are satisfied:

1. n π -covers m
2. any other $n' \in N_\mu$ that π -covers m also π -covers n

A node n π -covers a node m when n is an ancestor of m in the tree and the two nodes (their atoms) are comparable according to π . Starting from a node m and going upwards along a branch, the first comparable node encountered is its closest covering ancestor. Note that the notion of π -covering is not dependent on the measure functions in π , but solely on its classes.

Example 3.19 Reconsider the tree in Figure 2 and let π 's classes be as in Example 3.8. Then:

- Nodes 1, 2 and 4 do not have a covering ancestor.
- Nodes 3 and 5 are each covered by a single ancestor node, 1 and 4 respectively.
- Node 6, finally, is covered by both node 3 and node 1, the former of which is its closest covering ancestor.

Definition 3.20 An *atom abstraction operator (aao)* is a mapping abs from pairs of atoms to atoms, such that if A and A' are atoms, $abs(\{A, A'\})$ is an atom of which both A and A' are instances.

Let $\pi = ((C_1, \dots, C_n), (f_1, \dots, f_n))$ be a cm-pair, then an aao abs is called *π -safe* if the following both hold for any m-tree μ :

- if $n, n' \in N_\mu \cap C_j$ then any possible node m with $atom(m) = abs(\{atom(n), atom(n')\})$ must be in C_j
- if $n, n' \in N_\mu \cap C_j$ and $\neg(f_j(n) >_j f_j(n'))$ then either $abs(\{atom(n), atom(n')\}) = atom(n)$ or $f_j(n) >_j f_j(m)$ for any possible node m with $atom(m) = abs(\{atom(n), atom(n')\})$

Abusing notation, we occasionally write $abs(\{n, n'\})$ instead of $abs(\{atom(n), atom(n')\})$.

Example 3.21 Taking the most specific generalisation (msg) as abstraction operation can be safely combined with the use of functor measures (Definition 3.7), provided this is done with sufficient care. One possible way to proceed requires the following definition and lemma, (slightly) adapted from [12]:

Definition 3.22 Let $Term$ and $Atom$ denote the sets of terms and atoms, respectively, in a language \mathcal{L} . We define the function $s : Term \cup Atom \rightarrow \mathbb{N}$, counting the number of symbols in a term or an atom, by:

If $t = f(t_1, \dots, t_n), n > 0$
then $s(t) = 1 + s(t_1) + \dots + s(t_n)$
else $s(t) = 1$

Let the number of distinct variables in a term or atom t be $v(t)$. Define $h(t) = s(t) - v(t)$.

Note that $h(t) > 0$ for any non-variable t .

Lemma 3.23 If A and B are atoms such that B is strictly more general than A , then $h(A) > h(B)$.

It follows that taking the msg of two atoms is safe wrt cm-pairs with classes as in Example 3.5 and measure function f such that $f(A) = (f'(A), h(A)) \in \mathbb{N}^2, >_2$, where A is an atom, f' a functor measure (applicable to A) and $>_2$ the natural lexicographical order on \mathbb{N}^2 . A measure function of this kind will actually be used in Example 3.25 below.

We can now formulate a refined core algorithm for partial deduction. It is shown in Figure 3. To avoid some annoying technicalities, we assume that the goal to be partially deduced contains a single atom. This can always be achieved by adding a clause $goal \leftarrow G$ to the program. The algorithm is parameterised by a cm-pair $\pi = ((C_1, \dots, C_k), (f_1, \dots, f_k))$, a π -safe aao abs and an unfolding rule U . To obtain an executable partial deduction algorithm, these must be given concrete values.

Algorithm 3.24

Input

a program P and goal $\leftarrow A$

Output

a set of partially deduced atoms \mathcal{A}

Initialisation

$\mu_0 := \text{empty tree}$

$\mu_1 := \{(A, i_1)\}$

$i := 1$

While $\mu_{i-1} \neq \mu_i$ do

$\nu := ET(\mu_i, \{(l, \mathcal{A}_l) \mid l = (A_l, i_l) \in L_{\mu_i}^-, \mathcal{A}_l = BA(U(A_l, P))\}) \quad (*)$

$\mu_{i+1} := CT(\nu, L, N)$

where:

$L = \{l \in L_{\nu}^- \mid \exists 1 \leq j \leq k, \exists l' \in N_{\nu} : l \in C_j \ \& \ l' \ \pi\text{-covers } l$

$\ \& \ \neg(f_j(cca_{\pi}(\nu, l)) >_j f_j(l)) \ \& \ abs(\{l', l\}) = atom(l')\}$

$N = \{(cca_{\pi}(\nu, l), abs(\{cca_{\pi}(\nu, l), l\})) \mid \exists 1 \leq j \leq k : l \in C_j \cap L_{\nu}^- \ \& \ \exists l' \in N_{\nu} : l' = cca_{\pi}(\nu, l)$

$\ \& \ \neg(f_j(l') >_j f_j(l)) \ \& \ \neg \exists l'' \in N_{\nu} : l'' \ \pi\text{-covers } l \ \& \ abs(\{l'', l\}) = atom(l'')\}$

$i := i + 1$

Endwhile

$\mathcal{A} := \mathcal{A}_{\mu_i}$

Figure 3: An m-tree based algorithm template for partial deduction.

In essence, Algorithm 3.24 records the (to be) partially deduced atoms in an m-tree. Upon each iteration, it computes partial deductions (using U) for all the atoms in the tree that are not yet partially deduced (i.e. non-marked). It then adds the atoms in the corresponding resultants to the tree, as children of the atoms in whose partial deductions they appear. To guarantee termination, the weight of the new leaves is tested. If it is smaller than that of the most nearby comparable atom in the same branch, then the leaf is allowed to remain in the

tree. If not, then it is simply deleted if there is a more “abstract” atom in the same branch. If the latter also does not hold, then the whole subtree rooted in the most nearby comparable ancestor atom is removed, and its root is replaced by an atom more abstract than both the considered leaf and the removed root. The latter atom, of course, is to be partially deduced in the next iteration of the algorithm.

We illustrate the operation of Algorithm 3.24 on a simple example.

Example 3.25 Consider the program for reversing a list with the aid of an accumulating parameter:

$$\begin{aligned} rev([], Z, Z) &\leftarrow. \\ rev([X|Xs], Y, Z) &\leftarrow rev(Xs, [X|Y], Z). \end{aligned}$$

and the following query:

$$\leftarrow rev([1, 2|Xs], [], Z).$$

We will use Algorithm 3.24 with:

- a single class C_1 in π comprising all instances of $rev(X, Y, Z)$
- $f_1 = (|\cdot|_{rev, \{1\}}, h)$
- taking the msg as atom abstraction operator
- unfolding down to the first choice point, with a minimum of at least one level of unfolding (this determines U)

The m-trees constructed during partial deduction are depicted in Figure 4. For clarity, we have subscripted the ν -trees according to the μ_i they derive from. Node identifiers are not included, and the atoms in marked nodes are underlined>. Unmarked leaves are annotated by their weight under f_1 .

In the first iteration of the while-loop, the starting query is unfolded, using the second clause for rev until the call $rev(Xs, [2, 1], Z)$ is reached. At this point, both rev -clauses can be used to continue, so unfolding stops and $rev(Xs, [2, 1], Z)$ is added as an unmarked leaf to the m-tree, resulting in ν_1 . Since $f_1(rev([1, 2|Xs], [], Z)) = (2, 6) >_2 f_1(rev(Xs, [2, 1], Z)) = (0, 6)$, the fresh leaf is allowed to remain in the tree and therefore $\mu_2 = \nu_1$. Unfolding $rev(Xs, [2, 1], Z)$ one level results in two local descendent nodes, one of which is the empty goal. The atom in the other one is added to μ_2 , thus creating ν_2 . However, the cca of the leaf in ν_2 is the node with $rev(Xs, [2, 1], Z)$, which has a smaller weight under f_1 . Since the leaf atom is not an instance of any of its ancestor atoms, the subtree rooted in its cca is removed and replaced by a node with the msg of the two atoms involved. This gives us μ_3 . Note that indeed $f_1(rev(Xs, [2, 1], Z)) >_2 f_1(rev(Xs, [X, Y|Ys], Zs))$. Next, ν_3 is obtained similarly to ν_2 above. Again, the weight does not decrease properly, but this time the leaf atom is an instance of the atom in the leaf’s cca. μ_4 is therefore obtained by simply deleting ν_3 ’s leaf. Since now there are no unmarked leaves, a final iteration leaves the m-tree unchanged, and the algorithm halts with resulting set $\mathcal{A} = \{rev([1, 2|Xs], [], Z), rev(Xs, [X, Y|Ys], Zs)\}$. For the given problem, this result is very reasonable. While partial deduction as in [2] or [1] does not (globally) terminate on the present example, generalisation based on identical characteristic trees ([11], [20]) does ensure termination in this simple case. However, in general, that is not so, unless a depth bound is imposed on characteristic trees. Moreover, for some applications, wrapping together all calls with the same characteristic tree turns out to be too coarse a heuristic (as is illustrated by the example discussed in Section 3.3).

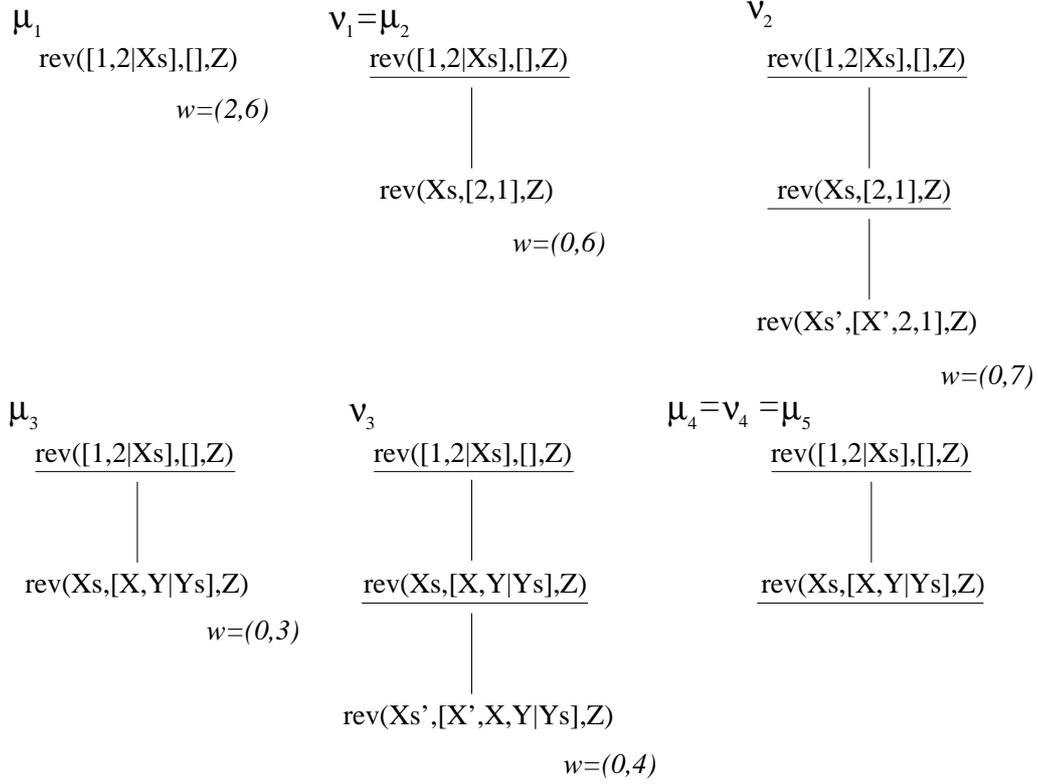


Figure 4: The m-trees for Example 3.25.

Let us now formally prove that (proper instances of) Algorithm 3.24 terminate(s). We set out with the following lemma:

Lemma 3.26 Every m-tree μ_i constructed while executing Algorithm 3.24, is subset-wise founded wrt π .

Proof Obviously, μ_0 and μ_1 are subset-wise founded wrt π . Now suppose μ_n is subset-wise founded wrt π and μ_{n+1} is constructed (i.e. $\mu_{n-1} \neq \mu_n$). We argue that also μ_{n+1} is subset-wise founded wrt π .

First, observe that the ν constructed from μ_n is not necessarily subset-wise founded. Indeed, it might contain freshly added, non-marked leaves where the corresponding measure function behaves non-monotonically. But it is exactly these leaves that are removed when constructing μ_{n+1} from ν . Either by simply deleting l when l is π -covered by some l' such that $abs(\{l', l\}) = atom(l')$, or by removing the terminal subtree rooted at its closest π -covering ancestor in ν when this is not so. In the latter case, the measure function is guaranteed to be monotonic on the node replacing $cca_\pi(\nu, l)$ in μ_{n+1} because abs is π -safe. \square

So, every $\mu_i \in \mathcal{M}_\pi$. We can take a further step:

Lemma 3.27 For every $i > 1$ such that an execution of Algorithm 3.24 defines μ_i , it holds that either $\mu_{i-1} >_\pi \mu_i$ or $\mu_{i-1} = \mu_i$.

Proof This result follows from combining Definition 3.14, Proposition 3.16, Definition 3.20 and Lemma 3.26. \square

Drawing together the major strands of Sections 3.1 and 3.2, our main result follows.

Theorem 3.28 Algorithm 3.24 terminates.

Proof This theorem is an immediate consequence of Theorem 3.15 and Lemma 3.27. \square

Finally, we also prove that Algorithm 3.24 is correct in the sense that it ensures closedness. (As noted in Section 2, independence can be obtained through post-execution renaming.)

Theorem 3.29 Let P be a program, input to Algorithm 3.24, and \mathcal{A} the set of atoms produced as output. Let P' be the partial deduction of \mathcal{A} in P using U . Then P' is \mathcal{A} -closed.

Proof First, a straightforward induction argument shows that in any m-tree constructed by Algorithm 3.24, only leaves can be unmarked. This implies that if $\mu_{i-1} = \mu_i$ then neither of the two has any unmarked nodes. (If there would be unmarked nodes in μ_{i-1} , then these would be leaves, and therefore either marked in or absent from μ_i .) It is therefore sufficient to show that for any marked node n in any μ_i ($i > 0$), the partial deduction of $atom(n)$ in P using U is \mathcal{A}_{μ_i} -closed.

Let n be a node in an m-tree μ , then we will denote as $C(n, \mu)$ the subset of N_μ consisting of the ancestors of n (including n) and its children (i.e. first level descendants). And we will denote as $\mathcal{A}_{C(n, \mu)}$ the set of atoms appearing in the nodes in $C(n, \mu)$. We will actually show that for any marked node n in any μ_i ($i > 0$), the partial deduction of $atom(n)$ in P using U is $\mathcal{A}_{C(n, \mu)}$ -closed. The proof proceeds through induction on i and has a trivial base case ($i = 1$). So, suppose that μ_{i+1} gets constructed and the property holds for all marked nodes in μ_i . Let ν be constructed from μ_i as in line (*) in the algorithm. Obviously, the property also holds for all marked nodes in ν . Now, in constructing μ_{i+1} from ν , no ancestor of a node is ever deleted while the node itself is kept, and a child of a node is only deleted if there is either an ancestor of this child (the child itself not included, and therefore also ancestor of the child's parent) with an at least equally general atom, or the child gets replaced by a node with an at least equally general atom. It follows that the desired property also holds for all marked nodes in μ_{i+1} . \square

Summarising, plugging suitable, concrete choices for π , abs and U into Algorithm 3.24 renders a partial deduction algorithm that, for any program P and goal $\leftarrow A$, is guaranteed to terminate in a non ad-hoc way, provides a flexible basis for polyvariant specialisation, and produces a specialised program (after some post-processing) that can be safely used with goals containing atoms that are instances (modulo post-execution renaming) of the atoms in its output \mathcal{A} .

To corroborate these claims, especially the one concerning polyvariance, in the next subsection, we turn our attention to some concrete instances of Algorithm 3.24, and briefly discuss their operation on a particularly interesting example.

3.3 A challenging example

Some well-known examples from the literature require polyvariance in order to achieve the best results, including the pattern-matching programs *match* ([9]) and *contains* ([18]). Good results for these can be reproduced in our framework, although global termination is not a significant problem in partial deduction of these programs.

In this subsection, we discuss an example for which both polyvariance and global termination are important ingredients. It is an example taken from [6] in which partial deduction is followed by an analysis using abstract interpretation. The analyser returns a single combined approximation for each predicate, and therefore the amount of polyvariance produced by partial deduction has an immediate effect on the precision of the analysis results. Note that an analyser could in principle produce several results for each predicate, in which case the problem of polyvariance would be shifted to the analyser, but this is not the point at issue in this example.

The following program is a simple implementation of a model elimination proof procedure (inspired by [29]). It was used in [6] as a basis for experiments on the specialisation of proof procedures.

$$\begin{aligned}
\textit{prove}((A, As), L) &\leftarrow \textit{prove}(A, L), \textit{prove}(As, L). \\
\textit{prove}(\textit{true}, _) &\leftarrow . \\
\textit{prove}(A, L) &\leftarrow \textit{member}(A, L). \\
\textit{prove}(A, L) &\leftarrow \textit{contrapositive}(A, B), \textit{neg}(A, A1), \textit{prove}(B, [A1|L]). \\
\textit{member}(X, [X|_]) &\leftarrow . \\
\textit{member}(X, [_|Xs]) &\leftarrow \textit{member}(X, Xs).
\end{aligned}$$

The procedure for *neg/2* is omitted here. The clauses of an object theory are added to the program in the form of unit clauses for *contrapositive/2*. A goal for the program typically has the form $\leftarrow \textit{prove}(A, [])$ where *A* is an atom to be proved. The specialisation problem is to derive a specialised version of the prover, given a particular object theory and a (partially known) atom to be proved.

The second argument of *prove* represents the “negative ancestor list” and contains the negations of the atoms that have been resolved with an object clause (in the fourth clause of the prover). It is initially empty, but grows longer as the proof proceeds. In general there may be infinite branches in the proof tree and therefore there is no bound on the length of the ancestor list in calls to *prove*. An infinite number of distinct calls to *prove/2* and hence to *member/2* (using the third clause) can arise in a computation.

We focus attention on the calls to *member/2*, which implements the “ancestor resolution” rule. As pointed out in [6], it is desirable that partial deduction generates sufficient different versions of *member/2* in order that further analysis can distinguish cases where ancestor resolution always fails from cases where it is possibly useful. In the basic algorithm for partial deduction, the problem is to find a plausible abstraction that allows enough versions to be generated. In [6] polyvariance was achieved by an ad-hoc technique.

We can contrast the treatment of *prove/2* and *member/2* in the basic algorithm and in the framework of Algorithm 3.24. Obviously some abstraction of atoms with these predicates is needed since an infinite number of different instances can arise in a computation. In the basic algorithm, the abstraction of *prove/2* and *member/2* is done independently. But using the framework of Algorithm 3.24 at least one version of *member/2* will be produced for each version of *prove/2* that is produced. (We assume that atoms containing *prove/2* and *member/2* are in different classes). This is because each call to *prove/2* has as ancestors only other calls to *prove/2*, but the first call to *member/2* on any branch is preceded by one or more calls to *prove/2*. Since abstraction takes place on each branch separately, the calls to *member/2* on separate branches will not be “confused”.

We argue that there is more “natural” polyvariance for *prove/2* than for *member/2* using typical abstractions (for example, using ideas based on characteristic paths ([11])). The problem with the basic algorithm is to ensure that sufficient polyvariance for *member/2* is produced. The framework of Algorithm 3.24 ensures that the natural polyvariance for *prove/2* is transferred to *member/2*.

Concrete instances of Algorithm 3.24 were implemented, modifying the *SP* system ([13]) to handle subset-wise founded marked trees. *SP* contains a number of different unfolding rules, which can be used in Algorithm 3.24. Most specific generalisation is used as the abstraction operator. For the class-measure pairs, we took various measures of the kinds introduced in Examples 3.5, 3.8 and 3.25. Different possibilities for the classes were tried. Classes can be simply based on the predicate names, or on some other factor that gives finer-grained distinctions, such as characteristic paths (with a fixed size bound).

There is some trade-off between the sophistication of the measure functions and the grain of the class distinctions. A constant measure function combined with a very fine-grained set of classes seems capable of producing many polyvariant versions, as does a sophisticated measure function with a simple set of classes. Much more experimentation is needed to find good heuristics for selection of unfolding rules, measure functions and class distinctions, but the framework of Algorithm 3.24 is rich enough to incorporate a wider range of strategies than previous proposals have allowed.

4 Discussion and Related Work

In the ensuing discussion, some of the partial deduction methods mentioned, treat normal logic programs, others only deal with definite ones. Although generalising the more elaborate restricted methods to normal logic programs seems often far from trivial, we will, in this section, gloss over this distinction. The cautious reader should therefore feel free to interpret what follows as pertaining to partial deduction of definite programs only.

A method for partial deduction of logic programs within the framework laid out by [23], was first presented in [2] (and further refined in [1], introducing dynamic renaming to provide additional opportunities for polyvariant specialisation). As does all currently published work in the setting of [23], it features set-wise global control (see Section 2). It concentrates on satisfying closedness and independence, and (global) termination is not addressed. Not surprisingly then, it is also not ensured: the proposed *abstract* operation fails to guarantee finiteness of \mathcal{A} . The latter is remedied in [26], but at a considerable price: except for the calls appearing in the starting goal, polyvariance is discarded. Combined with an eager unfolding rule, this still provides useful specialisation on a range of examples, but remains unsatisfactory as a general strategy (see [24]). The work in this paper provides the best of both worlds: flexible polyvariance and guaranteed termination.

Both Algorithms 2.8 and 3.24 are formulated in the style of the basic algorithm in [10], computing a set \mathcal{A} rather than (immediately) a partial deduction as in the above mentioned work. Doing so, independence can easily be ensured through post-execution renaming, and it is conceptually straightforward to add additional information to the manipulated atoms (e.g. the abstract substitutions in [10]’s enhanced algorithm). *SP* (see Section 3.3) actually uses various refinements of the above-mentioned basic algorithm. Whether applied in that context, or with methods immediately computing concrete partial deductions, the work in this paper

offers a useful setting for providing flexible global control, while ensuring termination.

Much of the (early) work in partial evaluation of logic programs, not within the framework established in [23] (see e.g. [9], [31]), while allowing polyvariance, addresses termination only casually, or relies on user provided annotations. (For a recent example of work within the latter strand, see [19].) However, approaches aiming at full automation, and relying on online methods, must include an automatic mechanism to prohibit non-termination. Two fairly recent systems of this kind are *Mixtus* ([32]) and *Paddy* ([30]). Typically, partial deduction that does not comply with the framework in [23], does not distinguish (explicitly) between local and global control, building one big evaluation structure that comprises both the local SLD-trees and something similar to our global m-tree. Restoring the use of a tree-like structure to the global control level in approaches based on [23] re-introduces fine-grained dependency registration at this level of consideration, while keeping the conceptual advantage of imposing a clear distinction between the two issues. It also renders the control strategy employed by other approaches more readily amenable to analysis within the “two level” context. Both *Mixtus* and *Paddy* can be understood to impose global control through the use of cm-pairs with “trivial” measure functions, in the sense that they map to well-founded sets where no element is larger than any other. As a result, every branch of the m-tree can for every class in the cm-pair contain just a single atom. As already pointed out at the end of Section 3.3, this is a valid approach if a fine-grained classification of atoms can be provided. *Mixtus*, for example, indeed does so, by imposing a refined control of unfolding at the local level and subsequently classifying atoms at the global level according to the local “may-loop” criterion. Its basic algorithm does not incorporate an atom abstraction operator; it ensures closedness by retaining clauses from the original program for atoms that do not get partially deduced. The “generalised restart” variant described in Section 2.5 of [32], however, does replace a partially deduced atom in the m-tree by the msg of a set of atoms including itself and some recursive descendant that has not been unfolded (and, somewhat surprisingly, possibly also some intermediate unfolded atoms (see [32])). Contrary to what is claimed in [32], this can cause non-termination: the resulting msg may be identical to the atom to be replaced by it. As in Algorithm 3.24, in such cases, no restart should be performed. Note that there is also no need to retain clauses from the original program: the recursive descendant is an instance of the partially deduced atom.

Partial evaluation algorithms developed for functional and procedural languages often make use of a so-called “binding-time analysis” (BTA). This technique was also used in the logic program specialiser *Logimix* ([15], [27]). The output of BTA is a description of different reachable program points (function calls or program statements) in which program variables are labeled as *static* or *dynamic*. Polyvariance is determined by the number of different versions of program points generated during BTA. Simpler BTA procedures fix the number of possible versions in advance. Finer-grained BTA can generate several versions of the same program point, but then the global termination problem appears as the problem of ensuring that BTA generates only a finite number of versions of program points. Thus although the termination problem is handled “offline”, it is essentially the same problem that we consider.

Chapter 14 of [15] introduces BTA based on the concept of “bounded static variation”. The BTA guarantees a finite, but potentially unlimited number of versions. The algorithm, which has apparently not yet been used in an actual partial evaluation system, essentially allows more variables to be classified as static than simpler BTA does. It does this by trying to identify variables that take on only a finite number of distinct values, though they may

appear to be dynamic. A variable is allowed to remain static so long as its concrete values are decreasing along a “dependency chain” which can be arbitrarily long. This approach is clearly related to our framework, in that termination is based on identifying arguments that do not decrease (with respect to some measure) along a computation path in an abstract execution of the program. It is stated in terms of one particular abstraction (static-dynamic descriptions) and given orderings of the program’s data values, whereas ours is flexible with regard to both the orderings (our measure functions) and the abstractions (the classes).

Termination of “online” partial evaluation in functional programming does not seem to have an accepted satisfactory solution. In the *Fuse* system ([38]), for example, the partial evaluator is allowed to loop in cases where the object program would loop. It is debatable whether this is satisfactory for functional programs, but in a logic programming context, where most recursive programs are bound to loop on certain (non-ground) call patterns, typically provided as input to the partial deduction process, it is clearly unacceptable.

The work on supercompilation ([37]), recently gaining renewed interest (see [33], [14] and further references there), seems quite closely related to partial deduction. Its basic generalisation strategy ([36]) is amenable to an analysis in terms of a bounded number of classes (“neighborhoods”) and (like *Mixtus* and *Paddy*) trivial measure functions, only admitting one term in each class in every branch of the structure built. (Supercompilation performs only one-step trivial unfoldings, and therefore lacks a local control level.) In fact, control of polyvariance in partial deduction by way of characteristic trees with a depth bound ([11], [13], [20]) can also be combined with the use of an m-tree in this way: one class per characteristic tree. (The neighbourhoods used in supercompilation reflect a common “computation history” of “order n ” (i.e. comprising exactly n steps). This seems to be related to the use of characteristic trees with depth bound n in logic programming.)

In all cases addressed above, we have not considered whether generalisation of atoms within a class returns an atom in the same class. Our present framework imposes this as a restriction (see Definition 3.20). However, with the necessary care, the framework can be generalised such that this limitation is lifted. We refer to future work.

Readers familiar with [4] will have noticed that, though drawing upon the same source of inspiration, the above development differs quite considerably from the approach to *local* control presented there. On the one hand, it is more complicated, due to the need for more complex manipulations of the tree structure. On the other hand, it is simpler, not needing an equivalent to the notion of a “hierarchical prefounding” (Definition 3.6 in [4]), essentially because (unlike selected atoms in SLD-trees) all nodes in a branch of an m-tree are recursive (or “proper” in the terminology introduced in [4]) descendants of all their tree ancestors. The latter phenomenon is also the reason why no “not to be measured class” is needed: calls to non-recursive (and therefore safe) predicates will never have an m-tree descendant with the same predicate symbol, anyway.

Finally, we briefly dwell upon an interesting “reduced deletion” variant² of Algorithm 3.24. Indeed, in cases where Algorithm 3.24 deletes (the subtree rooted in) the cca of a leaf and replaces it by the abstraction of the two nodes, one can alternatively just delete and replace the considered leaf. This is safe (termination-wise) if the abstraction operator satisfies Definition 3.20 and the $>_i$ are total orders (which will usually be the case). This reduced deletion variant provides even more opportunities for polyvariant specialisation than Algorithm 3.24

²First suggested by Michael Leuschel.

does. Moreover, since it only requires much simpler type 3 dtts, the relevant reformulation of Theorem 3.15 can probably be proved much more easily than is the case now. On the other hand, most existing approaches do delete partially deduced atoms more eagerly than this variant. The approach as it stands therefore provides a more suitable frame of reference for analysis and comparison of earlier work. It can be noted, by the way, that in such work (see e.g. [2], [1], [13], [26]), very often only roots of subtrees get deleted, the remaining nodes causing spurious definitions in the resulting partial deduction (subsequently removable in a “dead code deletion” phase). This is not surprising: it is exactly the introduction of an m-tree which makes the problem clearly visible.

5 Conclusion, Future Work

In this paper, we have investigated global control of partial deduction. We have identified the two central aspects to be addressed in this context: providing the right amount of polyvariance, while ensuring termination of the partial deduction in all cases. It is well-known that polyvariant specialisation can be a key factor in producing elegant, concise and, most important in this context, efficient specialised programs. However, the present state of the art in the field does not provide conclusive general rules or heuristics that might be able to guide this process such that optimal results are produced. Algorithms for partial deduction that leave open the details of control and abstraction (as opposed to detailed descriptions of particular partial deduction systems) are relatively rare, and our framework provides the most flexible such algorithm that we are aware of.

This paper reconsiders the global control issue, starting from the basic principles. Doing so, it develops a framework and an algorithm template that

- keep local and global control conceptually separated (this still allows for mutual interaction, if so desired, in practice)
- ensure global termination in a principled, non ad-hoc way
- nevertheless avoid imposing unnecessary a priori restrictions on the amount of possible polyvariance
- satisfy the closedness condition required for correctness of the specialised program

In this way, it provides both a conceptual framework in which various control strategies used in practical partial deduction (and evaluation) systems can be analysed, understood and compared, and an interesting starting point for further research, both conceptual and experimental, pursuing optimal control strategies for program specialisation.

Other topics for further research include:

- Ample experimentation with various concrete instances of Algorithm 3.24, or its reduced deletion variant, proposed in Section 4, will provide further insights into the merits, conceptual and practical, of the approach. In particular, the interaction of local and global control needs to be better understood.
- It seems very likely that a straightforward instance of Algorithm 3.24 (or, of course, a reduced deletion variant) will often produce *more* polyvariance than is actually desirable

(see Section 3.3). So, future research will have to determine criteria which can serve as a basis for further reducing and optimising polyvariance. Any such techniques can either directly be incorporated in Algorithm 3.24, or they can, perhaps more appropriately, be applied in a separate step, generating the specialised program, after running an analysis phase based on Algorithm 3.24. Obviously, both approaches can also be combined.

- At present, the user has to provide a cm-pair, to be used throughout the partial deduction process. This is no major drawback in an experimental context, and indeed, to a certain extent even an advantage: the user, likely to be an expert in the field, remains in control. However, in situations where specialisation is not the object of study, it is important to find techniques that allow automatic initialisation and refinement of cm-pairs depending on the particular specialisation problem at hand. An elaborate treatment of automatic initialisation and online refinement of measure functions of the “argument complexity weighing” kind (used to provide local control) can be found in [25]. An initial approach to automatic class refinement is also presented. However, global control provides a considerably more general setting, and much remains to be done. In the context of approaching optimality, offline analysis techniques (scrutinising the input to the partial deducer) seem promising to provide sensible initial values for cm-pairs, while online refinement can provide further fine-tuning.
- Finally, it will be interesting to study possible generalisations of the framework, both from a viewpoint of feasibility and usefulness. In particular, we have the following two issues in mind:
 - As pointed out above, the safety conditions on atom abstraction operators can be relaxed, no longer requiring the abstraction of two atoms to be in the same class as those atoms. However, type 3 dtts will then have to be redefined in such a way that termination is still guaranteed. It is also conceivable that two different sets of measure functions would be used: one to govern addition of leaves, and the second to control replacement of covering ancestors. All this seems perfectly possible, but further research will have to assess the technical details as well as the conceptual and practical gains resulting from such moves.
 - In [25], the requirement that (local control) measure functions be monotonic is relaxed to allow them not to be monotonic on a finite number of nodes in each branch, and it is shown how the use of such functions still guarantees termination. This is a point worthwhile pursuing also in the present context, since it for example dramatically improves the treatment of constants by measure functions based on argument complexity ([24], [25]).

Acknowledgements

We are grateful to Michael Leuschel who has carefully read a draft version of this paper and whose detailed and constructive comments have contributed to improvements of both contents and presentation. We also thank Danny De Schreye and André de Waal for stimulating discussions on the topic of this paper.

References

- [1] K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, 1993.
- [2] K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings NACLP'90*, pages 343–358, MIT Press, Austin, Texas, October 1990.
- [3] R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
- [4] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.
- [5] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings POPL'93*, ACM, Charleston, South Carolina, January 1993.
- [6] D. A. de Waal and J. P. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Proceedings CADE-12*, pages 207–221, Springer-Verlag, LNAI 814, Nancy, France, June/July 1994.
- [7] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
- [8] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.
- [9] H. Fujita. *An Algorithm for Partial Evaluation With Constraints*. Technical Report TR-258, ICOT, Tokyo, Japan, 1987.
- [10] J. Gallagher. Specialisation of logic programs: a tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, ACM Press, Copenhagen, June 1993.
- [11] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3&4):305–333, 1991.
- [12] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 229–244, Leuven, April 1990.
- [13] J. P. Gallagher. *A System for Specialising Logic Programs*. Technical Report TR-91-32, Computer Science Department, University of Bristol, U.K., November 1991.
- [14] R. Glück and M. H. Sørensen. Driving and partial deduction are equivalent. In M. Hermenegildo and J. Penjam, editors, *Proceedings PLILP'94*, pages 165–181, Springer-Verlag, LNCS 844, Madrid, Spain, 1994.
- [15] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

- [16] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 49–69, Springer-Verlag, LNCS 649, 1992.
- [17] J. Komorowski. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation*. PhD thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1981.
- [18] P. Kursawe. Pure partial evaluation and instantiation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 279–290, North-Holland, 1988.
- [19] M. Leuschel. Partial evaluation of the “real thing”. In F. Turini, editor, *Proceedings LOPSTR'94*, Springer-Verlag, Workshops in Computing Series, 1995.
- [20] M. Leuschel and D. De Schreye. *An Almost Perfect Abstraction Operator for Partial Deduction*. Technical Report CW199, Departement Computerwetenschappen, K.U.Leuven, Belgium, December 1994.
- [21] G. Levi and G. Sardu. Partial evaluation of metaprograms in a multiple worlds logic language. *New Generation Computing*, 6(2&3):227–247, 1988.
- [22] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
- [23] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3&4):217–242, 1991.
- [24] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, Departement Computerwetenschappen, K.U.Leuven, Belgium, February 1994.
- [25] B. Martens and D. De Schreye. *Advanced Techniques in Finite Unfolding*. Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, Belgium, October 1993. Submitted for publication.
- [26] B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1-2):97–117, 1994.
- [27] T. Mogensen and A. Bondorf. Logimix: a self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Proceedings LOPSTR'92*, pages 214–227, Springer-Verlag, Workshops in Computing Series, 1993.
- [28] A. Pettorossi and M. Proietti. Transformation of logic programs: foundations and techniques. *Journal of Logic Programming*, 19/20:261–320, 1994.
- [29] D. L. Poole and R. Goebel. Gracefully adding negation and disjunction to Prolog. In E. Shapiro, editor, *Proceedings ICLP'86*, pages 635–641, Springer-Verlag, LNCS 225, London, U.K., July 1986.
- [30] S. Prestwich. Online partial deduction of large programs. In *Proceedings PEPM'93*, pages 111–118, ACM, Copenhagen, Denmark, June 1993.

- [31] S. Safra. *Partial Evaluation of Concurrent Prolog and its Implications*. Technical Report Master's Thesis, CS86-24, Weizmann Institute, Department of Computer Science, 1986.
- [32] D. Sahlin. Mixtus: an automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
- [33] M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sanella, editor, *Proceedings ESOP'94*, pages 485–500, Springer-Verlag, LNCS 788, Edinburgh, U.K., April 1994.
- [34] L. Sterling and R. D. Beer. Meta interpreters for expert system construction. *Journal of Logic Programming*, 6(1&2):163–178, 1989.
- [35] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to metaprogramming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.
- [36] V. F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549, North-Holland, 1988.
- [37] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
- [38] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Proceedings of the 3rd ACM Conference on Functional Programming Languages and Computer Architecture*, pages 165–191, Springer-Verlag, LNCS 523, Cambridge, Massachusetts, USA, August 1991.