Efficiently Generating Efficient Generating Extensions in Prolog

Jesper Jørgensen* and Michael Leuschel**

K.U. Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: {jesper,michael}@cs.kuleuven.ac.be

Abstract. The so called "cogen approach" to program specialisation, writing a compiler generator instead of a specialiser, has been used with considerable success in partial evaluation of both functional and imperative languages.

This paper demonstrates that this approach is also applicable to partial evaluation of logic programming languages, also called partial deduction. Self-application has not been as much in focus in partial deduction as in partial evaluation of functional and imperative languages, and the attempts to self-apply partial deduction systems have, of yet, not been altogether that successful. So, especially for partial deduction, the cogen approach could prove to have a considerable importance when it comes to practical applications.

It is demonstrated that using the cogen approach one gets very efficient compiler generators which generate very efficient generating extensions which in turn yield (for some examples at least) very good and non-trivial specialisation.

1 Introduction

Partial evaluation has over the past decade received considerable attention both in functional (e.g. [23]), imperative (e.g. [1]) and logic programming (e.g. [13, 26, 42]). In the context of pure logic programs, partial evaluation is often referred to as *partial deduction*, the term partial evaluation being reserved for the treatment of impure logic programs. A convention we will also adhere to in this paper.

Guided by the *Futamura projections* (see e.g. [23]) a lot of effort, specially in the functional partial evaluation community, has been put into making systems self-applicable. A partial evaluation or deduction system is called *self-applicable* if it is able to effectively[-1] specialise itself. In that case one may, according to the second Futamura projection, obtain *compilers* from interpreters and, according to the third Futamura projection, a *compiler generator* (cogen for short).

[*] Supported by HCM Network "Logic Program Synthesis and Transformation".

[**] Supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation"

[-1] This implies some efficiency considerations, e.g. the system has to terminate within reasonable time constrains, using an appropriate amount of memory.

However writing an effectively self-applicable specialiser is a non-trivial task — the more features one uses in writing the specialiser the more complex the specialisation process becomes, because the specialiser then has to handle these features as well. This is why so far no partial evaluator for full Prolog (like MIXTUS [45], or PADDY [43]) has been made effectively self-applicable. On the other hand a partial deducer which specialises only purely declarative logic programs (like SAGE in [18] or the system in [7]) has itself to be written purely declaratively leading to slow systems and impractical compilers and compiler generators.

So far the only practical compilers and compiler generators have been obtained by striking a delicate balance between the expressivity of the underlying language and the ease with which it can be specialised. Two approaches for logic programming languages along this line are [11] and [39]. However the specialisation in [11] is incorrect with respect to some of the extra-logical built-ins, leading to incorrect compilers and compiler generators when attempting self-application (a problem mentioned in [7], see also [39, 28]). LOGIMIX [39] does not share this problem, but gives only modest speedups (when compared to results for functional programming languages, see [39]) when self-applied.

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 once a specialiser is given. Therefore, from a users point of view, whether a cogen is produced by self-application or not is of little importance, what is important is that it exists and that it has an improved performance over direct self-application. This is the background behind the approach to program specialisation called the *cogen approach*: instead of trying to write a partial evaluation system which is neither too inefficient nor too difficult to self-apply one simply writes a compiler generator directly. This is not as difficult as one might imagine at first sight: basically cogen turns out to be just a simple extension of a "binding-time analysis" for logic programs (something first discovered for functional languages in [21]).

In this paper we will describe the first cogen written in this way for a logic programming language: a small subset of Prolog.

The most noticeable advantages of the cogen approach is that the cogen and the compilers it generates can use all features of the implementation language. Therefore, no restrictions due to self-application have to be imposed (the compiler and the compiler generator don't have to be self-applied)! As we will see, this leads to extremely efficient compilers and compiler generators. So, in this case, having extra-logical features at our disposal makes the generation of compilers easier and less burdensome.

Some general advantages of the cogen approach are: the cogen manipulates only syntax trees and there is no need to implement a self-interpreter (meta-interpreter for the underlying language); values in the compilers are represented directly (there is no encoding overhead); and it becomes easier to demonstrate correctness for non-trivial languages (due to the simplicity of the transformation). In addition, the compilers are stand-alone programs that can be distributed without the cogen.

A further advantage of the cogen approach for logic languages is that the compilers and compiler generators can use the non-ground representation (and even a compiled version of it). This is in contrast to self-applicable partial deducers which *must* use the ground representation in order to be declarative (see [20, 34, 18]). In fact the non-ground representation executes several orders of magnitude faster than the ground representation (even after specialising, see [8]) and, as shown in [34], can be impossible to specialise satisfactorily by partial deduction alone. (Note that even [39] uses a "mixed" representation approach [31, 20]).

Although the Futamura projections focus on how to generate a compiler from an interpreter, the projections of course also apply when we replace the interpreter by some other program. In that case the program produced by the second Futamura projection is not called a compiler, but a *generating extension*. The program produced by the third Futamura projection could rightly be called a *generating extension generator* or gengen, but we will stick to the more conventional cogen.

The main contributions of this work are:

- the first description of a handwritten compiler generator (cogen) for a logic programming language which shows that such a program has quite an elegant and natural structure.
- a formal specification of the concept of *binding-time analysis* (BTA) in a (pure) logic programming setting and a description of how to obtain a generic algorithm for partial deduction from such a BTA (by describing how to obtain an unfolding and a generalisation strategy from the result of a BTA).
- benchmark results showing the efficiency of the cogen, the generating extensions and the specialised programs.

The paper is organised as follows: In Sect. 2 we formalise the concept of off-line partial deduction and the associated binding-time analysis. In Sect. 3 we present and explain our cogen approach in a pure logic programming setting. In Sect. 4 we present some examples and results underlining the efficiency of the cogen. We conclude with some discussions in Sect. 5.

2 Off-Line Partial Deduction

Throughout this paper, we suppose familiarity with basic notions in logic programming ([35]). Notational conventions are standard and self-evident. In particular, in programs, we denote variables through strings starting with (or usually just consisting of) an upper-case symbol, while the notations of constants, functions and predicates begin with a lower-case character.

We will also use the following not so common notations. Given a function $f : A \mapsto B$ we often use the *natural extension* of f, $f^* : 2^A \mapsto 2^B$, defined by $f^*(S) = \{f(s) \mid s \in S\}$. Similarly, given a function $f : A \mapsto 2^B$ we also define the function $f_\cup : 2^A \mapsto 2^B$, by $f_\cup(S) = \cup_{s \in S} f(s)$. Both f^* and $f_\cup$ are homomorphisms[0] from 2^A to 2^B. Given a function $f : A \times B \mapsto C$ and an element $a \in A$ we define the curried version of f, $f_a : B \mapsto C$, by $f_a(X) = f(a, X)$. Finally, we will denote by $A_{\text{if}} \rightarrow A_{\text{then}}; A_{\text{else}}$ the Prolog conditional.

2.1 A Generic Partial Deduction Method

Given a logic program P and a goal G, *partial deduction* produces a new program P' which is P "specialised" to the goal G; the aim being that the specialised program P' is more efficient than the original program P for all goals which are instances of G.

The underlying technique of partial deduction is to construct "incomplete" SLDNF-trees and then extract the specialised program P' from these incomplete search trees (by taking resultants, see below). An *incomplete* SLDNF-tree is a SLDNF-tree which, in addition to success and failure leaves, may also contain leaves where no literal has been selected for a further derivation step. Leaves of the latter kind will be called *dangling* ([37]). In the context of partial deduction these incomplete SLDNF-trees are obtained by applying an unfolding rule, defined as follows.

Definition 1. *(Unfolding rule) An* unfolding rule *U is a function which, given a program P and a goal G, returns a finite, (possibly) incomplete and non-trivial*[1] *SLDNF-tree for $P \cup \{G\}$.*

Given an incomplete SLDNF-tree, partial deduction will generate a set of clauses by taking resultants. Resultants are defined as follows.

Definition 2. *($resultants(\tau)$, $leaves(\tau)$) Let P be a normal program and A an atom. Let τ be a finite, incomplete SLDNF-tree for $P \cup \{\leftarrow A\}$ in which A has been selected in the root node. Let $\leftarrow G_1, \ldots, \leftarrow G_n$ be the goals in the (non-root) leaves of the non-failing branches of τ. Let $\theta_1, \ldots, \theta_n$ be the computed answers of the derivations from $\leftarrow A$ to $\leftarrow G_1, \ldots, \leftarrow G_n$ respectively. Then the set of resultants, $resultants(\tau)$, is defined to be the set of clauses $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$. We also define the set of leaves, $leaves(\tau)$, to be the atoms occurring in the goals $G_1, \ldots, G_n$.*

Partial deduction, as defined for instance in [36] or [4], uses the resultants for a given set of atoms $\mathbf{A}$ to construct the specialised program (and for each atom in $\mathbf{A}$ a different specialised predicate definition will be generated). Under the conditions stated in [36], namely closedness and independence, correctness of the specialised program is guaranteed.

In a lot of practical approaches (e.g. [12, 13, 15, 31, 28, 29]) independence is ensured by using a *renaming* transformation which maps dependent atoms

[0] The function $h : 2^A$ to 2^B is a homomorphism iff $h(\emptyset) = \emptyset$ and $h(S \cup S') = h(S) \cup h(S')$.

[1] A trivial SLDNF-tree is one whose root is a dangling leaf. This restriction is necessary to obtain correct partial deductions. See also Definition 2 below.

to new predicate symbols. Adapted correctness results can be found in [3] (see also [32]). Renaming is often combined with argument *filtering* to improve the efficiency of the specialised program (see e.g. [14]).

Closedness can be ensured by using the following outline of a partial deduction algorithm (similar to the ones used in e.g. [12, 13, 29, 30]).

Algorithm 1 *(Partial deduction)*

1. *Let S_0 be the set of atoms to be specialised and let $i = 0$.*
2. *Apply the unfolding rule U to each element of S_i: $\Gamma_i = U^*_P(S_i)$.*
3. *$S_{i+1} = abstract(S_i \cup leaves_{\cup}(\Gamma_i))$*
4. *If $S_{i+1} \neq S_i$ (modulo variable renaming) increment i and restart at step 2, otherwise generate the specialised program by applying a renaming (and filtering) transformation to $resultants_{\cup}(\Gamma_i)$.*

The abstraction operation is usually used to ensure termination and can be formally defined as follows ([12, 13]).

Definition 3. *An operation $abstract(S)$ is any operation satisfying the following conditions. Let S be a finite set of atoms; then $abstract(S)$ is a finite set of atoms S' with the same predicates as those in S, such that every atom in S is an instance of an atom in S'.*

If the above algorithm terminates then the closedness condition is satisfied. Finally note that in the above algorithm the atoms in $leaves_{\cup}(\Gamma_i)$ are all added and abstracted simultaneously, i.e. the algorithm progresses in a breadth-first manner. In general this will yield a different result from a depth-first progression (i.e. adding one atom at a time). If however *abstract* is a homomorphism[2] then both progressions will yield exactly the same set of atoms and thus the same specialisation.

2.2 Off-Line Partial Deduction and Binding-Time Analysis

In Algorithm 1 one can distinguish between two different levels of control. The unfolding rule U controls the construction of the incomplete SLDNF-trees. This is called the *local control* (we will use the terminology of [13, 38]). The abstraction operation controls the construction of the set of atoms for which local SLDNF-trees are built. We will refer to this aspect as the *global control*.

The control problems have been tackled from two different angles: the so-called *off-line* versus *on-line* approaches. The *on-line* approach performs all the control decisions *during* the actual specialisation phase (in our case the one depicted in Algorithm 1). The *off-line* approach on the other hand performs an analysis phase *prior* to the actual specialisation phase, based on some rough descriptions of what kinds of specialisations will have to be performed. The analysis phase provides annotations which then guide the control aspect of the proper specialisation phase, often to the point of making it completely trivial.

[2] I.e. $abstract(\emptyset) = \emptyset$ and $abstract(S \cup S') = abstract(S) \cup abstract(S')$.

Partial evaluation of functional programs ([10, 23]) has mainly stressed off-line approaches, while supercompilation of functional ([47, 46]) and partial deduction of logic programs ([15, 45, 6, 9, 37, 38, 29, 33]) have concentrated on on-line control. (Some exceptions are [39, 31, 28].)

The main reason for using the off-line approach is to achieve effective self-application ([24]). But the off-line approach is in general also more efficient, since many decisions concerning control are made before and not during specialisation. For the cogen approach to be efficient it is vital to use the off-line approach, since then the (local) control can be hard-wired into the generating extension.

Most off-line approaches perform a so called *binding-time analysis (BTA)* prior to the specialisation phase. This phase classifies arguments to predicate calls as either *static* or *dynamic*. The value of a static argument is definitely known (bound) at specialisation time whereas a dynamic argument is not definitely known (it might only be known at the actual run-time of the program). In the context of partial deduction, a static argument can be seen as being a term which is guaranteed not to be more instantiated at run-time (it can never be less instantiated at run-time). For example if we specialise a program for all instances of $p(a, X)$ then the first argument to p is static while the second one is dynamic — actual run-time instances might be $p(a, b), p(a, Z), p(a, X)$ but not $p(b, c)$. We will also say that an atom is static if all its arguments are static and likewise that a goal is static if it consist only of static (literals) atoms.

We will now formalise the concept of a binding-time analysis. For that we first define the concept of divisions which classify arguments into static and dynamic ones.

Definition 4. *(Division) A* division of arity n *is a couple* (S, D) *of sets of integers such that* $S \cup D = \{1, \ldots, n\}$ *and* $S \cap D = \emptyset$.

We also define the function divide which, given a division and a tuple of arguments, divides the arguments into the static and the dynamic ones: $divide_{(S,D)}((t_1, \ldots, t_n)) = ((t_{i_1}, \ldots, t_{i_k}), (t_{j_1}, \ldots, t_{j_l}))$ *where* $(i_1, \ldots, i_k)$ *(resp.* $(j_1, \ldots, j_k)$*) are the elements of* S *(resp.* D*) in ascending order.*

As a notational convenience we will use $(\delta_1, \ldots, \delta_n)$ to denote a division (S, D) of arity n, where $\delta_i = s$ if $i \in S$ and $\delta_i = d$ if $i \in D$. For example (s, d) denotes the division $(\{1\}, \{2\})$ of arity 2. From now on we will also use the notation $Pred(P)$ to denote the predicate symbols occurring inside a program P. We now define a division for a program P which divides the arguments of every predicate $p \in Pred(P)$ into the static and the dynamic ones:

Definition 5. *(Division for a program) A* division Δ for a program P *is a mapping from* $Pred(P)$ *to divisions having the arity of the corresponding predicates. In accordance with the notations outlined at the beginning of this section, we will often write* Δ_p *for* $\Delta(p)$. *We also define the function* Δ_p^s *by* $\Delta_p^s(\overline{x}) = \overline{y}$ *iff* $divide_{\Delta_p}(\overline{x}) = (\overline{y}, \overline{z})$. *Similarly we define the function* Δ_p^d *by* $\Delta_p^d(\overline{x}) = \overline{z}$ *iff* $divide_{\Delta_p}(\overline{x}) = (\overline{y}, \overline{z})$.

Example 1. $(\{1\},\{2\})$ is a division of arity 2 and $(\{2,3\},\{1\})$ a division of arity 3 and we have for instance $divide_{(\{2,3\},\{1\})}((a,b,c)) = ((b,c),(a))$. Let P be a program containing the predicate symbols $p/2$ and $q/3$. Then $\Delta = \{p/2 \mapsto (\{1\},\{2\}), q/3 \mapsto (\{2,3\},\{1\})\}$ is a division for P (using the notational convenience introduced above we can also write $\Delta = \{p/2 \mapsto (s,d), q/3 \mapsto (d,s,s)\}$). We then have for example $\Delta_q^s((a,b,c)) = (b,c)$ and $\Delta_q^d((a,b,c)) = (a)$.

Divisions can be ordered. A division is more general than another one if it classifies more arguments as dynamic. This is captured by the following definition.

Definition 6. *(Partial order of divisions) Divisions of the same arity are partially ordered:* $(S,D) \sqsubseteq (S',D')$ *iff* $D \subseteq D'$.

We also define the notation $\bot_n = (\{1,\ldots,n\},\emptyset)$ *and* $\top_n = (\emptyset,\{1,\ldots,n\})$.

This order can be extended to divisions for some program P. *We say that* Δ' *is* more general *than* Δ, *denoted by* $\Delta \sqsubseteq \Delta'$, *iff for all predicates* $p \in Pred(P)$: $\Delta_p \sqsubseteq \Delta'_p$.

As already mentioned, a binding-time analysis will, given a program P (and some description of how P will be specialised), perform a pre-processing analysis and return a *division* for P describing when values will be bound (i.e. known). It will also return an *annotation* which will then guide the local unfolding process of the actual partial deduction. From a theoretical viewpoint an annotation restricts the possible unfolding rules that can be used (e.g. the annotation could state that predicate calls to p should never be unfolded whereas calls to q should always be unfolded). We therefore define annotations as follows:

Definition 7. *(Annotation) An* annotation $\mathcal{A}$ *is a set of unfolding rules (i.e. it is a subset of the set of all possible unfolding rules).*

In order to be really off-line, the unfolding rules in the annotation should not take the unfolding history into account and should not depend "too much" on the actual values of the static (nor dynamic) arguments. In the following subsection we will come back on what annotations can look like from a practical viewpoint. We are now in a position to formally define a binding-time analysis in the context of (pure) logic programs:

Definition 8. *(BTA,BTC) A* binding-time analysis *(BTA) yields, given a program* P *and an initial division* Δ_0 *for* P, *a couple* $(\mathcal{A},\Delta)$ *consisting of an annotation* $\mathcal{A}$ *and a division* Δ *for* P *more general than* Δ_0. *We will call the result of a binding-time analysis a* binding-time classification *(BTC).*

The initial division Δ_0 gives information about how the program will be specialised. In fact Δ_0 specifies what the initial atom(s) to be specialised (i.e. the ones in S_0 of Algorithm 1) will look like (if p' does not occur in S_0 we simply set $\Delta_0(p') = \bot_n$). The role of Δ is to give information about what the atoms in Algorithm 1 will look like at the global level. In that light, not all BTC as specified above are correct and we now develop a safety criterion for a BTC wrt a given program. Basically a BTC is safe iff every atom that can potentially appear in one of the sets S_i of Algorithm 1 (given the restrictions imposed by the annotation of the BTA) corresponds to the patterns described by Δ. Note that if a predicate p is always unfolded by the unfolding rule used in Algorithm 1 then it is irrelevant what the value of Δ_p is.

For simplicity, we will from now on impose that a *static* argument must be *ground*.[3] In particular this guarantees our earlier requirement that the argument will not be more instantiated at run-time.

Definition 9. *(*safe wrt Δ*) Let P be a program and let Δ be a division for P and let $p(\bar{t})$ be an atom with $p \in Pred(P)$. Then $p(\bar{t})$ is* safe wrt Δ *iff $\Delta_p^s(\bar{t})$ is a tuple of ground terms. A set of atoms S is* safe wrt Δ *iff every atom in S is safe wrt Δ. Also a goal G is* safe wrt Δ *iff all the atoms occurring in G are safe wrt Δ.*

For example $p(a, X)$ is safe wrt $\Delta = \{p/2 \mapsto (s, d)\}$ while $p(X, a)$ is not.

Definition 10. *(safe BTC, safe BTA) Let $\beta = (\mathcal{A}, \Delta)$ be a BTC for a program P and let $U \in \mathcal{A}$ be an unfolding rule. Then β is a* safe BTC for P and U *iff for every goal G, which is safe wrt Δ, U returns an incomplete SLDNF-tree whose leaf goals are safe wrt Δ. Also β is a* safe BTC for P *iff it is a safe BTC for P and for every unfolding rule $U \in \mathcal{A}$. A BTA is* safe *if for any program P it produces a safe BTC for P.*

So, the above definition requires atoms to be safe in the leaves of incomplete SLDNF-trees, i.e. at the point where the atoms get abstracted and then lifted to the *global* level.[4] So, in order for the above condition to ensure safety at all stages of Algorithm 1, the particular abstraction operation should not abstract atoms which are safe wrt Δ into atoms which are no longer safe wrt Δ. This motivates the following definition:

Definition 11. *An abstraction operation abstract is* safe wrt a division Δ *iff for every finite set of atoms S, which is safe wrt Δ, $abstract(S)$ is also safe wrt Δ.*

2.3 A Particular Off-Line Partial Deduction Method

In this subsection we define a specific off-line partial deduction method which will serve as the basis for the cogen developed in the remainder of this paper. For simplicity, we will from now on restrict ourselves to definite programs. Negation will in practice be treated in the cogen either as a built-in or via the *if-then-else* construct (see Appendix A).

Let us first define a particular unfolding rule.

[3] This simplifies stating the safety criterion of a BTA because one does not have to reason about "freeness". In a similar vein this also makes the BTA itself easier.

[4] Also, when leaving the pure logic programming context and allowing extra-logical built-ins (like `=../2`) a *local* safety condition will also be required.

Definition 12. *($U_{\mathcal{L}}$) Let $\mathcal{L} \subseteq Pred(P)$. We will call $\mathcal{L}$ the set of* reducible *predicates. Also an atom will be called reducible iff its predicate symbol is in $\mathcal{L}$. We then define the unfolding rule $U_{\mathcal{L}}$ to be the unfolding rule which selects the leftmost reducible atom in each goal (and of course, for atomic goals $\leftarrow A$ in the root, it always selects A).*

We will use such unfolding rules in Algorithm 1 and we will restrict ourselves (to avoid distracting from the essential points) to safe BTA's which return results of the form $\beta = (\{U_{\mathcal{L}}\}, \Delta)$. In the actual implementation of the cogen (Appendix B) we use a slightly more liberal approach in the sense that specific program points (calls to predicates) are annotated as either reducible or non-reducible. Also note that nothing prevents a BTA from having a pre-processing phase which splits the predicates according to their different uses.

Example 2. Let P be the following program

(1) $p(X) \leftarrow q(X,Y), q(Y,Z)$
(2) $q(a,b) \leftarrow$
(3) $q(b,a) \leftarrow$

Let $\Delta = \{p \mapsto (s), q \mapsto (s,d)\}$. Then $\beta = (\{U_{\{q\}}\}, \Delta)$ is a safe BTC for P. For example the goal $\leftarrow p(a)$ is safe wrt Δ and unfolding it according to $U_{\{q\}}$ will lead (via the intermediate goals $\leftarrow q(a,Y), q(Y,Z)$ and $\leftarrow q(b,Z)$) to the empty goal $\Box$ which is safe wrt Δ. Note that every selected atom is safe wrt Δ.[5] Also note that $\beta' = (\{U_{\{\}}\}, \Delta)$ is a *not* a safe BTC for P. For instance, for the goal $\leftarrow p(a)$ the unfolding rule $U_{\{\}}$ just performs one unfolding step and thus stops at the goal $\leftarrow q(a,Y), q(Y,Z)$ which contains the unsafe atom $q(Y,Z)$.

The only thing that is missing in order to arrive at a concrete instance of Algorithm 1 is a (safe) abstraction operation, which we define in the following.

Definition 13. *(gen_{Δ}, $abstract_{\Delta}$) Let P be a program and Δ be a division for P. Let $A = p(\bar{t})$ with $p \in Pred(P)$. We then denote by $gen_{\Delta}(A)$ an atom obtained from A by replacing all dynamic arguments of A (according to Δ_p) by distinct variables.*
We also define the abstraction operation $abstract_{\Delta}$ to be the natural extension of the function gen_{Δ}: $abstract_{\Delta} = gen^*_{\Delta}$.

For example, if $\Delta = \{p/2 \mapsto (s,d), q/3 \mapsto (d,s,s)\}$ then $gen_{\Delta}(p(a,b)) = p(a,X)$ and $gen_{\Delta}(q(a,b,c)) = q(X,b,c)$. Then $abstract_{\Delta}(\{p(a,b), q(a,b,c)\}) = \{p(a,X), q(X,b,c)\}$. Note that, trivially, $abstract_{\Delta}$ is safe wrt Δ.

Note that $abstract_{\Delta}$ is a homomorphism and hence, as already noted, we can use a depth-first progression in Algorithm 1 and still get the same specialisation. This is something which we will actually do in the practical implementation.

In the remainder of this paper we will use the following off-line partial deduction method:

[5] As already mentioned, this is not required in definition 10 but (among others) such a condition will have to be incorporated for the selection of extra-logical built-in's.

Algorithm 2 *(off-line partial deduction)*

1. *Perform a BTA (possibly by hand) returning results of the form* $(\{U_{\mathcal{L}}\}, \Delta)$
2. *Perform Algorithm 1 with* $U_{\mathcal{L}}$ *as unfolding rule and* $abstract_{\Delta}$ *as abstraction operation. The initial set of atoms* S_0 *should only contain atoms which are safe wrt* Δ.

Proposition 1. *Let* $(\{U_{\mathcal{L}}\}, \Delta)$ *be a safe BTC for a program P. Let S_0 be a set of atoms safe wrt Δ. If Algorithm 2 terminates then the final set S_i only contains atoms safe wrt Δ.*

We will explain how this particular partial deduction method works by looking at an example.

Example 3. We use a small generic parser for a set of languages which are defined by grammars of the form $S ::= aS|X$ (where X is a placeholder for a terminal symbol). The example is adapted from [26] and the parser P is depicted in Fig. 1.

Given the initial division $\Delta_0 = \{nont/3 \mapsto (s,d,d), t/3 \mapsto \bot_3\}$ a BTA might return the following result $\beta = (\{U_{\{t/3\}}\}, \Delta)$ where $\Delta = \{nont/3 \mapsto (s,d,d), t/3 \mapsto (s,d,d)\}$. It can be seen that β is a safe BTC for P.

Let us now perform the proper partial deduction for $S_0 = \{nont(c,R,T)\}$. Note that the atom $nont(c,R,T)$ is safe wrt Δ_0 (and hence also wrt Δ). Unfolding the atom in S_0 yields the SLD-tree in Fig. 2. We see that the atoms in the leaves are $\{nont(c,V,T)\}$ and we obtain $S_1 = S_0$. The specialised program after renaming and filtering looks like:

$nont_c([a|V],R) \leftarrow nont_c(V,R)$
$nont_c([c|R],R) \leftarrow$

(1)	$nont(X,T,R) \leftarrow\ t(a,T,V), nont(X,V,R)$
(2)	$nont(X,T,R) \leftarrow t(X,T,R)$
(3)	$t(X,[X|ES],ES) \leftarrow$

Fig. 1. A parser

3 The cogen approach for logic programming languages

For presentation purposes we from now on suppose that in Algorithm 2 the initial set S_0 consists of just a single atom A_0 (a convention adhered to by a lot of practical partial deduction systems).

A *generating extension* of a program P with respect to a given safe BTC $(\{U_{\mathcal{L}}\}, \Delta)$ for P, is a program that performs specialisation (using part 2 of Algorithm 2) of any atom A_0 which is safe wrt Δ. So in the case of the parser

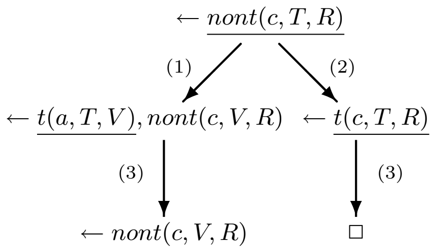

Fig. 2. Unfolding the parser of Fig. 1

from Ex. 3 a generating extension is a program that, when given the safe call $nont(c, R, T)$, produces the residual program shown in the example.

A *compiler generator*, *cogen*, is a program that given a program P and a safe $BTC\ \beta$ for P produces a generating extension of P wrt β.

We will first consider what the generating extensions wrt a program P and a safe $BTC\ \beta$ should look like. Once this is clear we will consider what *cogen* should look like.

As already stated, a generating extension should specialise safe calls to predicates. Let us first consider the unfolding aspect of specialisation. The partial deduction algorithm first unfolds the initial top-level atom (to ensure a non-trivial tree). It then proceeds with the unfolding until no more reducible atoms can be selected and collects the atoms in the leaves of the unfolded SLDNF-tree. This process is repeated for all the new (generalised) atoms which have not yet been unfolded, until no more new non-reducible atoms are found. Notice that all predicates may potentially have to be unfolded.

The crucial idea for simplicity and efficiency of the generating extension is to incorporate a specific predicate p_u for each predicate p/n. This predicate has $n + 1$ arguments and is tailored towards unfolding calls to p/n. The first n arguments correspond to the arguments of the call to p/n which has to be unfolded. The last argument collects the result of the unfolding process. More precisely, $p_u(t_1, ..., t_n, B)$ will succeed for each branch of the incomplete SLDNF-tree obtained by applying the unfolding $U_{\mathcal{L}}$ to $p(t_1, ..., t_n)$ whereby it will return in B the atoms in the leaf of the branch[6] and also instantiate $t_1, ..., t_n$ via the composition of *mgu*'s of the branch. For complete SLDNF-trees (i.e. for atoms which get fully unfolded) the above can be obtained very *efficiently* by simply executing the original predicate definition of p for the goal $\leftarrow p(t_1, ..., t_n)$ (no atoms in the leaves have to be returned because there are none). To handle the case of incomplete SLDNF-trees we just have to adapt the definition of p so that unfolding can be stopped (for non-reducible predicates according to $U_{\mathcal{L}}$) and so that in that case the atoms in the leaves are collected.

[6] For reasons of clarity and simplicity in unflattened form.

This can be obtained very easily by transforming every clause defining the predicate p/n into a clause for $p_u/(n+1)$, as done in the following definition. The following could actually be called a *compiled* non-ground representation, and contributes much to the final efficiency of the generating extensions.

Definition 14. *Let P be a program and $C = p(\bar{t}) \leftarrow A_1, ..., A_k$ a clause of P defining a predicate symbol p/n. Let $\mathcal{L} \subseteq Pred(P)$ be a set of reducible predicate symbols. We then define the clause $C_u^{\mathcal{L}}$ for the predicate p_u to be:*

$$p_u(\bar{t}, [\mathcal{R}_1, ..., \mathcal{R}_k]) \leftarrow \mathcal{S}_1, ..., \mathcal{S}_k$$

where

1. $\mathcal{S}_i$ *$= q_u(\overline{s}, \mathcal{R}_i)$ and $\mathcal{R}_i$ is a fresh unused variable, if $A_i = q(\overline{s})$ is reducible*
2. $\mathcal{S}_i$ *$= true$ and $\mathcal{R}_i = A_i$, if A_i is not reducible*

We will denote by $P_u^{\mathcal{L}}$ the program obtained by applying the above transformation to every clause in P and removing all true atoms from the bodies.

In the above definition inserting a literal of the form $q_u(\overline{s}, \mathcal{R}_i)$ corresponds to further unfolding whereas inserting *true* corresponds to stopping the unfolding process. In the case of Ex. 3 with $\mathcal{L} = \{t/3\}$, applying the above to the program P of Fig. 1 gives rise to the following program $P_u^{\mathcal{L}}$:

```
nont_u(X,T,R,[V1,nont(X,V,R)]) :- t_u(a,T,V,V1).
nont_u(X,T,R,[V1]) :- t_u(X,T,R,V1).
t_u(X,[X|R],R,[]).
```

Evaluating the above code for the call `nont_u(c,T,R,Leaves)` yields two computed answers which correspond to the two branches in Fig. 1:

```
> ?-nont_u(c,T,R,Leaves).
  T = [a | _52]
  Leaves = [[],nont(c,_52,R)]
Yes ;
  T = [c | R]
  Leaves = [[]]
Yes
```

The above code is of course still incomplete as it only handles the unfolding process and we have to extend it to treat the global level as well. Firstly, calling p_u only returns the atoms of one leaf of the SLDNF-tree, so we need to add some code that collects the information from all the leaves. This can be done very efficiently using Prolog's `findall` predicate. So in the following call `findall(B,nont_u(c,R,T,B),Bs)` the `Bs` will be instantiated to the following list `q[[[],nont(c,_48,_49)],[[]]]` which essentially corresponds to the leaves of the SLDNF-tree in Fig. 2, since by flattening out we obtain: `[nont(c,_48,_49)]`. Furthermore, if we call

```
findall(clause(nont(c,T,R),Bdy),nont_u(c,T,R,Bdy),Cs)
```

we will even get in `Cs` a representation of the two resultants of Ex. 3.

Once all the resultants have been generated, the body atoms have to be generalised (using gen_Δ) and unfolded if they have not been encountered yet. The easiest way to achieve this is to add a function p_m for each non-reducible predicate such that, p_m implements the global control aspect of the specialisation. That is, for every atom $p(\bar{t})$, if one calls $p_m(\bar{t}, R)$ then R will be instantiated to the residual call of $p(\bar{t})$ (i.e. the call after filtering and renaming, for instance the residual call of $p(a, b, X)$ might be $p_1(X)$). At the same time p_m also generalises this call, checks if it has been encountered before and if not, unfolds the atom, generates code and prints the resultants (residual code) of the atom. We have the following definition of p_m:

Definition 15. *Let P be a program and p/n be a predicate defined in P. Let $\mathcal{L} \subseteq Pred(P)$ be a set of reducible predicate symbols. For $p \in Pred(P)$ we define the clause C^p_m, defining the predicate p_m, to be:*

```
$p_m(\bar{t}, R) \leftarrow$
    $(find\_pattern(p(\bar{t}), R) \rightarrow true$
     $;\ (insert\_pattern(p(\bar{s}), H),$
        $findall(C, (p_u(\bar{s}, B), treat\_clause(H, B, C)), Cs),$
        $pp(Cs),$
        $find\_pattern(p(\bar{t}), R)))$.
```

where $p(\bar{s}) = gen_\Delta(p(\bar{t}))$. Finally we define $P^{\mathcal{L}}_m = \{C^p_m \mid p \in Pred(P) \setminus \mathcal{L}\}$.[7]

In the above, the predicate *find_pattern* checks whether its first argument is a call that has been encountered before and its second argument is the residual call to this (with renaming and filtering performed). This is achieved by keeping a list of the predicates that have been encountered before along with their renamed and filtered calls. So, if the call to *find_pattern* succeeds, then R has been instantiated to the residual call of $p(\bar{t})$, if not, then the other branch of the conditional is tried.

The predicate *insert_pattern* will add a new atom (its first argument) to the list of atoms encountered before and return (in its second argument H) the generalised, renamed and filtered version of the atom. The atom H will provide (maybe further instantiated) the head of the resultants to be constructed. This call to *insert_pattern* is put first to ensure that an atom is not specialised over and over again at the global level.

The call to $findall(C, (p_u(\bar{s}, B), treat\_clause(H, B, C)), Cs)$ unfolds the generalised atom $p(\bar{s})$ and returns a list of residual clauses for $p(\bar{s})$ (in Cs). The call to $p_u(\bar{s}, B)$ inside *findall* returns a leaf goal of the SLDNF-tree for $p(\bar{s})$. This goal is going to be the body of a residual clause with head H. For each of the atoms in the body of this clause two things have to be done. First, for each atom a

[7] This corresponds to saying that only reducible atoms can occur at the global level, and hence only reducible atoms can be put into the initial set of atoms S_0 of Algorithm 1. If this is not what you want then just change the above definition to "$p \in Pred(P)$" or to "$p \in (Pred(P) \setminus \mathcal{L}) \cup \{p_0\}$".

specialised residual version has to be generated if necessary. Second, each atom has to be replaced by a call to a corresponding residual version. Both of these tasks can be performed by calling the corresponding "m" function of the atoms, so if a body contains an atom $p(\bar{t})$ then $p_m(\bar{t}, R)$ is called and the atom is replaced by the value of R. The task of treating the body in this way is done by the predicate *treat_clause* and the third argument of this is the new clauses.

The predicate *pp* pretty-prints the clauses of the residual program. The last call to *find_pattern* will instantiate R to the residual call of the atom $p(\bar{t})$.

We can now define what a generating extension of a program is:

Definition 16. *Let P be a program, $\mathcal{L} \in Pred(P)$ a set of predicates and $(\{U_\mathcal{L}\}, \Delta)$ a safe BTC for P, then the* generating extension *of P with respect to $(\{U_\mathcal{L}\}, \Delta)$ is the program $P_g = P_u^\mathcal{L} \cup P_m^\mathcal{L}$.*

The complete generating extension for Ex. 3 is shown in Fig. 3.

```
nont_m(B,C,D,E) :-
   (find_pattern(nont(B,C,D),E) -> true
    ; (insert_pattern(nont(B,F,G),H),
         findall(I,(nont_u(B,F,G,J),treat_clause(H,J,I)),K),
         pp(K),
         find_pattern(nont(B,C,D),E)
         )).
nont_u(B,C,D,[E,memo(nont(B,G,D))]) :- t_u(a,C,G,E).
nont_u(H,I,J,[K]) :- t_u(H,I,J,K).
t_u(L,[L|M],M,[]).
```

Fig. 3. The generating extension for the parser

The generating extension is called as follows: if one wants to specialise an atom $p(\bar{t})$, where p is one of the non-reducible predicates of the subject program P then one calls the predicate p_m of the generating extension in the following way $p_m(\bar{t},\_)$.

The job of the cogen is now quite simple: given a program P and a safe BTC β for P, generate a generating extension for P consisting of the two parts described above. The code of the essential parts of our cogen is shown in Appendix B. The predicate `predicate` generates the definition of the global control m-predicates for each non-reducible predicate of the program whereas the predicates `clause`, `bodys` and `body` take care of translating clauses of the original predicate into clauses of the local control u-predicates. Note how the second argument of `bodys` and `body` corresponds to code of the generating extension whereas the third argument corresponds to code produced at the next level, i.e. at the level of the specialised program. Further details on extending the *cogen* to handle built-ins and the if-then-else can be found in Appendix A.

4 Examples and Results

In this section we present some experiments with our *cogen* system as well as with some other specialisation systems. We will use three example programs to that effect.

The first program is the parser from Ex. 3. We will use the same annotation as in the previous sections: $nont \mapsto (s, d, d)$.

The second example program is the "mixed" meta-interpreter (sometimes called *InstanceDemo*) for the ground representation of [12, 13, 31] in which the goals are "lifted" to the non-ground representation for resolution. We will specialise this program given the annotation $solve \mapsto (s, d)$, i.e. we suppose that the object program is given and the query to the object program is dynamic.

Finally we also experimented with a regular expression parser, which tests whether a given string can be generated by a given regular expression. The example is taken from [39]. In the experiment we used $dgenerate \mapsto (s, d)$ for the initial division, i.e. the regular expression is fully known whereas the string is dynamic.

4.1 Experiments with COGEN

The Tables 1, 2 and 3 summarise our benchmarks of the COGEN system. The timings were by using Prolog by BIM on a Sparc Classic running Solaris (timings, at least for Table 1, were almost identical for a Sun 4).

Program	Time	Annotation
parser	0.02 s	$nont \mapsto (s, d, d)$
solve	0.06 s	$solve \mapsto (s, d)$
regexp	0.02 s	$dgenerate \mapsto (s, d)$

Table 1. Running COGEN

Program	Time	Query
parser	0.01 s	$nont(c, T, R)$
solve	0.01 s	$solve(''\{q(X) \leftarrow p(X), p(a) \leftarrow\}'', Q)$
regexp	0.03 s	$dgenerate(''(a + b) * .a.a.b'', S)$

Table 2. Running the generating extension

Program	Speedup Factor	Runtime Query
parser	2.35	$nont(c, [\overbrace{a, \ldots, a}^{18}, c, b], [b])$
solve	7.23	$solve(''\{q(X) \leftarrow p(X), p(a) \leftarrow\}'', '' \leftarrow q(a)'')$
regexp	101.1	$dgenerate(''(a + b) * .a.a.b'', ''abaaaabbaab'')$

Table 3. Running the specialised program

The results depicted in Tables 1, 2 and 3 are very satisfactory. The generating extensions are generated very efficiently and also run very efficiently. Furthermore the specialised programs are also very efficient and the speedups are very satisfactory. The specialisation for the *parser* example corresponds to the one obtained in Ex. 3. By specialising *solve* our system cogen was able to remove almost all the overhead of the ground representation, something which has been achieved for the first time in [12]. In fact, the specialised program looks like this:

```
solve__0([]).
solve__0([struct(q,[B])|C]) :- solve__0([struct(p,[B])]), solve__0(C).
solve__0([struct(p,[struct(a,[])])|D]) :- solve__0([]), solve__0(D).
```

The specialised program obtained for the *regexp* example actually corresponds to a deterministic automaton, a feat that has also been achieved by the system logimix in [39]. For further details about these examples, as well as the experiments, see [25].

4.2 Experiments with other Systems

We also performed the experiments using some other specialisation systems. All systems were able to satisfactorily handle the *parser* example and came up with (almost) the same specialised program as cogen. More specific information is presented in the following paragraphs.

mixtus ([45]) is a partial evaluator for full Prolog which is not (effectively) self-applicable. We experimented with version 0.3.3 of mixtus. mixtus came up with exactly the same specialisation as our cogen for the *parser* and *solve* examples. mixtus was also able to specialise the *regexp* program, but not to the extent of generating a deterministic automaton.

We experimented with the sp system (see [12]), a specialiser for a subset of Prolog (not including the *if-then-else*). For the *solve* example sp was able to obtain the same specialisation as cogen, but only after re-specialising the specialised program a second time. Due to the heavy usage of the *if-then-else* the *regexp* example could not be handled directly by sp.

logimix ([39]) is a self-applicable partial evaluator for a subset of Prolog, containing *if-then-else*, side-effects and some built-in's. This system falls within the off-line setting and requires a binding time annotation. It is not (yet) fully automatic, in the sense that the program has to be hand-annotated. For the *parser* and *regexp* examples, logimix came up with almost the same programs than cogen. We were not able to annotate *solve* properly. It might be that this example cannot be handled by logimix because the restrictions on the annotations are more severe than ours (in cogen the unfoldable predicates do not require a division and cogen allows non-deterministic unfolding — the latter seems to be crucial for the *solve* example).

leupel ([28, 31]) is a (not yet effectively self-applicable) partial evaluator for a subset of Prolog, very similar to the one treated by logimix. The system is guided by an annotation phase which is unfortunately also not automatic. The annotations are “semi-online”, in the sense that conditions (tested in an on-line manner) can be given on when to make a call reducible or non-reducible. For the *parser* and *regexp* examples the system performed the same specialisation as

COGEN. For the *solve* example LEUPEL even came up with a better specialisation than COGEN, in the sense that unfolding has also been performed at the object level:

```
solve__1([]).
solve__1([struct(q,[struct(a,[])])|A]) :- solve__1(A).
solve__1([struct(p,[struct(a,[])])|A]) :- solve__1(A).
```

Such optimisations depend on the particular object program and are therefore outside the reach of purely off-line methods.

CHTREE is a fully automatic system for a declarative subset of Prolog (similar to the language handled by SP) based on the work in [29, 33]. It is an on-line system which has a very precise abstraction operation, minimising specialisation losses. We used a local unfolding rule based on the homeomorphic embedding relation (see e.g. [33, 46]). For the *solve* example the CHTREE came up with a better specialisation than COGEN, almost identical to the one obtained by LEUPEL (but this time fully automatically). Due to the heavy usage of the *if-then-else* the *regexp* example could, similarily to SP, not be handled directly by CHTREE.

We also did some experiments with the PADDY system (see [43]) written for full Eclipse (a variant of Prolog). PADDY basically performed the same specialisation of *solve* as CHTREE or LEUPEL, but left some useless tests and clauses inside. PADDY was also able to specialise the *regexp* program, but again not to the extent of generating a deterministic automaton.

Finally we tried out the self-applicable partial deducer SAGE (see [18]) for the logic programming language Gödel. SAGE came up with (almost) the same specialised program for the *parser* example as COGEN. SAGE performed little specialisation on the *solve* example, returning almost the unspecialised program back. Due to the heavy usage of the *if-then-else* the *regexp* example could not be handled by SAGE.

4.3 Comparing Transformation Times

The systems which gave us access to the transformation times were PADDY, MIXTUS, LEUPEL, CHTREE and LOGIMIX. The results can be found in Table 4. The columns marked by *spec* contain the times needed to produce the specialised program (i.e. the time to perform the first Futamura projection), whereas the columns marked by *genex* contain the times needed to produce the generating extensions (i.e. performing the second Futamura projection). The latter columns only make sense for COGEN, for the self-applicable system LOGIMIX as well as for $\text{COGEN}_{\text{logimix}}$ obtained via the third Futamura projection of LOGIMIX. As can be seen in Table 4, COGEN is by far the fastest system overall, as well for specialisation as for compiler generation, while producing almost the best specialised code. More details about the experiments can be found in [25]. Note however that the timings of CHTREE include the printing of tracing information and that a rather naive implementation of the homeomorphic embedding relation was used.

Finally the figures in Tables 1 and 2 really shine when compared to the compiler generator and the generating extensions produced by the self-applicable SAGE system. Unfortunately self-applying SAGE is currently not possible for normal users, so we had to take the timings from [18]: generating the compiler generator takes about 100 hours (including garbage collection), generating a generating extension took for the examples (which are probably more complex than the ones treated in this section) in [18] at least 11.8 hours (with garbage collection). The speedups by using the generating extension instead of the partial evaluator range from 2.7 to 3.6 but the execution times for the system (including pre- and post-processing) still range from 113s to 447s.

Specialiser	Prolog System	Architecture	*parser* *genex*	*parser* *spec*	*solve* *genex*	*solve* *spec*	*regexp* *genex*	*regexp* *spec*
COGEN	BIM	Sparc Classic	0.02 s	0.01 s	0.06 s	0.01 s	0.02 s	0.03 s
MIXTUS	SICStus	Sparc Classic	-	0.14 s	-	1.36 s	-	13.63 s
PADDY	Eclipse	Sun4	-	0.05 s	-	0.80 s	-	3.17 s
CHTREE	BIM	Sparc Classic	-	0.21 s	-	9.07 s	-	-
LEUPEL	BIM	Sparc Classic	-	0.11 s	-	0.64 s	-	4.00 s
LOGIMIX	SICStus	Sparc Classic	1.47 s	0.02 s	-	-	1.28 s	0.09 s
$\text{COGEN}_{\text{logimix}}$	SICStus	Sparc Classic	1.10 s	0.02 s	-	-	0.98 s	0.08 s

Table 4. Comparative Table of Specialisation Times

5 Discussion and Future Work

In comparison to other partial deduction methods the cogen approach may, at least from the examples given in this paper, seem to do quite well with respect to speedup and quality of residual code, and outperform any other system with respect to transformation speed. But this efficiency has a price. Since our approach is off-line it will of course suffer from the same deficiencies than other off-line systems when compared to on-line systems. Also, no partially static structures were needed in the above examples and our system cannot handle these, so it will probably have difficulties with something like the *transpose* program (see [12]) or with a non-ground meta-interpreter. However, our notion of BTA and BTC is quite a coarse one and corresponds roughly to that used in early work on self-applicability of partial evaluators for functional programming languages, so one might expect that this could be refined considerably.

Although our approach is closely related to the one for functional programming languages there are still some important differences. Since computation in our cogen is based on unification, a variable is not forced to have a fixed binding time assigned to it. In fact the binding-time analysis is only required to be safe, and this does not enforce this restriction. Consider the following program:

```
g(X) :- p(X),q(X)
```

```
p(a).
q(a).
```

If the initial division Δ_0 states that the argument to `g` is dynamic, then Δ_0 is safe for the program and the unfolding rule that unfolds predicates `p` and `q`. The residual program that one gets by running the generating extensions is:

```
g__0(a).
```

In contrast to this any cogen for a functional language known to us will classify the variable `X` in the following analogue functional program (here exemplified in Scheme) as dynamic:

```
(define (g X) (and (equal? X a) (equal? X a)))
```

and the residual program would be identical to the original program.

One could say that our system allows divisions that are not uniformly congruent in the sense of Launchbury [27] and essentially, our system performs specialisation that a partial evaluation system for a functional language would need some form of *driving* to be able to do.

Whether application of the cogen approach is feasible for specialisation of other logical programming languages than Prolog is hard to say, but it seems essential that such languages have some metalevel built-in predicates, like Prolog's `findall` and `call` predicates, for the method to be efficient. This means that it is probably not possible to use the approach (efficiently) for Gödel. Further work will be needed to establish this.

Related Work in Partial Evaluation

The first hand-written compiler generator based on partial evaluation principles was, in all probability, the system *RedCompile* for a dialect of Lisp [2]. Since then successful compiler generators have been written for many different languages and language paradigms [44, 21, 22, 5, 1, 16].

In the context of definite clause grammars and parsers based on them, the idea of hand writing the compiler generator has also been used in [40, 41].[8] However it is not based on (off-line) partial deduction. The exact relationship to our work is currently being investigated.

Future Work

The most obvious goal of the near future is to see if a complete and precise binding-time analysis can be developed. Since we imposed that a *static* term must be ground, one might think that the *BTA* corresponds exactly to groundness analysis. This is however not entirely true because a standard groundness analysis gives information about the arguments at the point where a call is

[8] Thanks to Ulrich Neumerkel for pointing this out.

selected (and often imposing left-to-right selection). In other words, it gives groundness information at the local level when using some standard execution. A BTA however requires groundness information about the arguments of calls in the leaves, i.e. at the point where these atoms are lifted to the global level. So what we actually need is a groundness analysis adapted for unfolding rules and not for standard execution of logic programs. However, by re-using and running a standard groundness analysis on a transformed version of the program to be specialised, we can come up with a reasonable BTA. More details, along with some initial experiments using the PLAI system [19], can be found in [25].

On a slightly longer term one might try to extend the cogen and the binding-time analysis to handle partially static structures. It also seems natural to investigate to what extent more powerful control and specialisation techniques (like the unfold/fold transformations, [42]) can be incorporated into the cogen in the context of conjunctive partial deduction ([32, 17]).

Acknowledgements

We thank Maurice Bruynooghe, Bart Demoen, Danny De Schreye, André De Waal, Robert Glück, Gerda Janssens, Bern Martens, Torben Mogensen and Ulrich Neumerkel for interesting discussions on this work. We also thank André De Waal for helping us with some partial evaluation experiments, Gerda Janssens for providing us with valuable information about abstract interpretation and Bern Martens for finding the title of the paper and for providing us with formulations for the text of the paper. Bern Martens also provided valuable feedback on a draft of this paper. Finally we are grateful to Danny De Schreye for his stimulating support and to anonymous referees for their helpful comments.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. L. Beckman, A. Haraldson, Ö. Oskarsson, and E. Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.
3. K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, October 1993.
4. K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 343–358. MIT Press, 1990.
5. L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings*, volume 844 of *LNCS*, pages 198–214, Madrid, Spain, 1994. Springer-Verlag.
6. R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.

7. A. Bondorf, F. Frauendorf, and M. Richter. An experiment in automatic self-applicable partial evaluation of prolog. Technical Report 335, Lehrstuhl Informatik V, University of Dortmund, 1990.
8. A. F. Bowers and C. A. Gurr. Towards fast and declarative meta-programming. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 137–166. MIT Press, 1995.
9. 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.
10. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of POPL'93*, Charleston, South Carolina, January 1993. ACM Press.
11. H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.
12. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
13. J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
14. J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.
15. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
16. R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, LNCS 982, pages 259–278. Springer-Verlag, 1995.
17. R. Glück, J. Jørgensen, B. Martens, and M. Sørensen. Controlling conjunctive partial deduction of definite logic programs. Technical Report CW 226, Departement Computerwetenschappen, K.U. Leuven, Belgium, February 1996. Submitted.
18. C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.
19. M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(4):349–366, 1992.
20. P. Hill and J. Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, 1994. To be published in *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5*. Oxford Science Publications, Oxford University Press.
21. C. K. Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.
22. C. K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. Working paper, 1992.
23. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
24. N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
25. J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. Technical Report CW 221, K.U. Leuven, Belgium, February 1996. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

26. J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 49–69. Springer-Verlag, LNCS 649, 1992.
27. J. Launchbury. *Projection Factorisations in Partial Evaluation.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.
28. M. Leuschel. Partial evaluation of the "real thing". In L. Fribourg and F. Turini, editors, Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, LNCS 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.
29. M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 1–16, Utrecht, Netherlands, September 1995. Springer-Verlag.
30. M. Leuschel and D. De Schreye. An almost perfect abstraction operation for partial deduction using characteristic trees. Technical Report CW 215, K.U. Leuven, Belgium, October 1995. Submitted for Publication. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
31. M. Leuschel and D. De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 253–263, La Jolla, California, June 1995. ACM Press.
32. M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. Technical Report CW 225, Departement Computerwetenschappen, K.U. Leuven, Belgium, February 1996. Submitted.
33. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In *this volume*.
34. M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 495–509, Portland, USA, December 1995. MIT Press. Extended version as Technical Report CW 210, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
35. J. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.
36. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
37. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 1995. To Appear.
38. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.
39. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.
40. G. Neumann. Transforming interpreters into compilers by goal classification. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 205–217, Leuven, Belgium, 1990.
41. G. Neumann. A simple transformation from Prolog-written metalevel interpreters into compilers and its implementation. In A. Voronkov, editor, Logic Programming. *Proceedings of the First and Second Russian Conference on Logic Programming*, LNCS 592, pages 349–360. Springer-Verlag, 1991.
42. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19 & 20:261–320, May 1994.

43. S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.
44. S. A. Romanenko. 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, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.
45. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
46. M. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, December 1995. MIT Press.
47. V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

A Extending the cogen

It is straightforward to extend the cogen to handle primitives, i.e. built-ins (`=`/2, `not`/1, `=..`/2, `call`/1,...) or externally defined user predicates. The code of these predicates will not be available and therefore no predicates to unfold them can be generated. The generating extension can either contain code that completely evaluates calls to primitives in which case the call will then be marked reducible or code that produces residual calls to such predicates in which case the call is marked non-reducible. So we extend the transformation of Def. 14 with the following two rules:

3. $\mathcal{S}_i = A_i$ and $\mathcal{R}_i = []$ if A_i is a reducible built-in
4. $\mathcal{S}_i = \mathit{true}$ and $\mathcal{R}_i = A_i$ if A_i is a non-reducible built-in

As a last example of how to extend the method we will show how to handle the Prolog version of the conditional: $A_{\text{if}} \rightarrow A_{\text{then}}; A_{\text{else}}$. For this we will introduce the notation $G^{\mathcal{R}}$ where $G = A_1, ..., A_k$ to mean the following:

$$G^{\mathcal{R}} = \mathcal{S}_1, ..., \mathcal{S}_k$$

where $\mathcal{S}_i, \mathcal{R}_i$ are defined as in Def. 14 and $\mathcal{R} = [\mathcal{R}_1, ..., \mathcal{R}_k]$ (i.e. this allows us perform the transformations recursively on the sub-components of a conditional).

If the test of a conditional is marked as reducible then the generating extension will simply contain a conditional with the test unchanged and where the two "branches" contain code for unfolding the two branches (similar to the body of a function indexed by "u"), i.e. Def. 14 is extended with the following rule:

5. $\mathcal{S}_i = (G_1 \rightarrow (G_2^{\mathcal{R}}, \mathcal{R}_i = \mathcal{R}); (G_3^{\mathcal{R}'}, \mathcal{R}_i = \mathcal{R}'))$ and $\mathcal{R}_i$ is a fresh variable, if $A_i = (G_1 \rightarrow G_2 \;; G_3)$ is reducible.

If the test goal of the conditional is non-reducible then we assume that the three subgoals are either a call to a non-reducible predicate, a call to a non-reducible (dynamic) primitive or another dynamic conditional. This restriction is not severe, since if a program contains conditionals that get classified as dynamic by the BTA and these contain arbitrary subgoals then the program may by a simple source language transformation be transformed into a program which satisfies the restriction. Def. 14 is extended with the following rule:

6. $S_i = (A'_1, A'_2, A'_3)^{[\mathcal{R},\mathcal{R}',\mathcal{R}'']}$ and $\mathcal{R}_i = (\mathcal{R} \rightarrow \mathcal{R}'; \mathcal{R}'')$, if $A_i = (A'_1 \rightarrow A'_2; A'_3)$ is non-reducible.

where A'_1, A'_2 and A'_3 are goals that satisfy the restriction above. This restriction ensures that the three goals $\{A'_i \mid i = 1, 2, 3\}$ compute their residual code independently of each other and the residual code for the conditional is then a conditional composed from this code.

B A Prolog cogen

This appendix contains the listing of the cogen. The system is available via `http://www.cs.kuleuven.ac.be/~lpai`.

```
/* ----------- */
/*  C O G E N  */
/* ----------- */

cogen :-
  findall(C,predicate(C),Clauses1),
  findall(C,clause(C),Clauses2),
  pp(Clauses1),
  pp(Clauses2).

flush_cogen :- print_header,flush_pp.

predicate(clause(Head,[if([find_pattern(Call,V)],[true],
                          [insert_pattern(GCall,H),
                           findall(NClause,
                                  (RCall,treat_clause(H,Body,NClause)),
                                   NClauses),
                           pp(NClauses),
                           find_pattern(Call,V)])])) :-
  generalise(Call,GCall),
  add_extra_argument("_u",GCall,Body,RCall),
  add_extra_argument("_m",Call,V,Head).


clause(clause(ResCall,ResBody)) :-
  ann_clause(Call,Body),
  add_extra_argument("_u",Call,Vars,ResCall),
  bodys(Body,ResBody,Vars).
bodys([],[],[]).
bodys([G|GS],GRes,VRes) :-
  body(G,G1,V),
  filter_cons(G1,GS1,GRes,true),
  filter_cons(V,VS,VRes,[]),
  bodys(GS,GS1,VS).

filter_cons(H,T,HT,FVal) :-
        ((nonvar(H),H = FVal) -> (HT = T) ; (HT = [H|T])).

body(unfold(Call),ResCall,V) :-
  add_extra_argument("_u",Call,V,ResCall).
body(memo(Call),true,memo(Call)).
body(call(Call),Call,[]).
body(rescall(Call),true,rescall(Call)).
body(if(G1,G2,G3),    /* Static if: */
     if(RG1,[RG2,(V=VS2)],[RG3,(V=VS3)]),V) :-
  bodys(G1,RG1,VS1), bodys(G2,RG2,VS2), bodys(G3,RG3,VS3).
body(resif(G1,G2,G3), /* Dynamic if: */
    [RG1,RG2,RG3],if(VS1,VS2,VS3)) :-
  body(G1,RG1,VS1), body(G2,RG2,VS2), body(G3,RG3,VS3).

generalise(Call,GCall) :-
  delta(Call,STerms,_), Call =.. [Pred|_],
  delta(GCall,STerms,_), GCall =.. [Pred|_].

add_extra_argument(T,Call,V,ResCall) :-
  Call =.. [Pred|Args],res_name(T,Pred,ResPred),
  append(Args,[V],NewArgs),ResCall =.. [ResPred|NewArgs].

res_name(T,Pred,ResPred) :-
  name(PE_Sep,T),string_concatenate(Pred,PE_Sep,ResPred).

print_header :-
  print(’/’),print(’*  --------------------  *’),print(’/’),nl,
  print(’/’),print(’*  GENERATING EXTENSION  *’),print(’/’),nl,
  print(’/’),print(’*  --------------------  *’),print(’/’),nl,
  print(’:’),print(’- reconsult(memo).’),nl,
  print(’:’),print(’- reconsult(pp).’),nl,
  (static_consult(List) -> pp_consults(List) ; true),nl.
```