# Efficiently Generating Efficient Generating Extensions in Prolog

Jesper Jørgensen and Michael Leuschel

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

e-mail: {jesper,michael}@cs.kuleuven.ac.be

Tel.: ++32(0)16-32{7560,7555}

Fax: ++32(0)16-327996

**Abstract**

The so called "cogen approach", writing a compiler generator instead of a specialiser, to program specialisation 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 succesful. 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 very good and non-trivial specialisation.

**Keywords:** partial deduction, off-line specialization, cogen approach

# 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", writing a compiler generator instead of a specialiser, to program specialisation 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 succesful. 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 very good and non-trivial specialisation.

## 1 Introduction

*Partial evaluation* has over the past decade received considerable attention both in functional (e.g. [24]) and logic programming (e.g. [14, 26, 39]). 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 unpure logic programs. A convention we will also adhere to in this paper.

Guided by the *Futamura projections* a lot of effort, specially in the functional partial evaluation community, has been put on making systems self-applicable. A partial evaluation or deduction system is called *self-applicable* if it is able to specialise itself. In case a system is self-applicable 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).

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

* Supported by HCM Network "Logic Program Synthesis and Transformation".
** Supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation"

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 [42], or Paddy [40]) 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 inefficient, 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 [12] and [38]. However the specialisation in [12] 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 [38, 28]). The partial evaluator Logimix of [38] does not share this problem, but only modest speedups (when compared to results for functional programming languages, see the remarks in [38]) are obtained when performing self-application.

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 of a partial evaluator or deduction system 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 without being forced to strictly preserve the operational semantics when specialising. 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 generator manipulates only syntax trees (no need to implement a self-interpreter); values in the compilers (generating extensions) 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, generating extensions 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. This is in contrast to self-applicable partial deducers which *must* use the ground representation in order to be declarative (see [20, 33, 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 [33], can be impossible to specialise satisfactorily by partial deduction alone.[3] (Note that even [38] uses a "mixed" representation approach which lies in between the ground and non-ground style ).

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 with any program. In this 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.

## 2 Off-Line Partial Deduction

Throughout this paper, we suppose familiarity with basic notions in logic programming ([34]). 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 $f^*(S) = \{f(s) \mid s \in S\}$. Similarily, 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 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 `If` $\rightarrow$ `Then ; Else` the Prolog conditional.

[3] It is a matter of future research to see whether a self-applicable specialiser can be written using the new implementation and specialisation scheme of the ground representation developed in [33].

### 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. 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 an incomplete SLDNF-tree for $P \cup \{G\}$. In addition, for every atomic goal $\leftarrow A$, it returns an SLDNF-tree $\tau$ in which the atom $A$ is selected in the root node.[4]

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 $\tau$ 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 $\tau$. 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 [35] 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 [35], namely closedness and independence, correctness of the specialised program is guaranteed.

In a lot of practical approaches (e.g. [13, 14, 16, 31, 28, 29]) independence is ensured by using a *renaming* transformation which maps dependent atoms to new predicate symbols. Adapted correctness results can be found in [3]. Renaming is often combined with argument filtering to improve the efficiency of the specialised program (see e.g. [15]).

Closedness can be ensured by using the following outline of a partial deduction algorithm (similar to the ones used in e.g. [13, 14, 29, 30]).

[4] This restriction is necessary to obtain correct partial deductions. Also see the definition of resultants below.

**Algorithm 2.1** *(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.

**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.

### 2.2 Off-Line Partial Deduction and Binding-Time Analysis

In algorithm 2.1 one can distinguish 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 [14, 37]). The abstraction operation controls the construction of the set of atoms for which local SLDNF-trees are build. 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 2.1). The *off-line* approach on the other hand performs an analysis phase prior to the actual specialisation phase. 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.

Partial evaluation of functional programs ([10, 24]) has mainly stressed the former, while supercompilation of functional ([44, 43]) and partial deduction of logic programs ([16, 42, 6, 9, 36, 37, 29, 32]) have concentrated on on-line control. (Some exceptions are [38, 31, 28].)

The main reason for using the off-line approach has been to be able to efficiently perform self-application ([25]), 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 what is called a *binding-time analysis (BTA)* prior to the specialisation phase. This phase classifies arguments to predicate calls as either *static* or *dynamic*. A static argument is one the value of which 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 neither more nor less bound at run-time. We will 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 divide arguments into static and dynamic ones. First we define divisions for tuples of arguments.

**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 $i \in D$. From now on we will also use the notation $Pred(P)$ to denote the predicate symbols occurring inside a program $P$. A division for a program $P$ divides the arguments of every predicate of $p \in Pred(P)$ into the static and the dynamic ones:

**Definition 5.** (Division for a program)
A *division* $\Delta$ *for a program* $P$ is a mapping from $Pred(P)$ to divisions having the arity of the corresponding predicates. We will often write $\Delta_p$ for $\Delta(p)$. We also define the function $\Delta_p^s$ by $\Delta_p^s(x) = y$ iff $divide_{\Delta_p}(x) = (y, z)$. Similarily we define the function $\Delta_p^d$ by $\Delta_p^d(x) = z$ iff $divide_{\Delta_p}(x) = (y, z)$.

*Example 1.* $(\{1\}, \{2\})$ is a division of arity 2 and $(\{2, 3\}, \{1\})$ a division of arity 3 and we have for instance $divide_{(\{2,3\},\{1\})}((a, b, X)) = ((b, X), (a))$. 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 $\Delta_q^s((a, b, X)) = (b, X)$ and $\Delta_q^d((a, b, X)) = (a)$.

Divisions can be ordered, the intuition being that the more arguments that are classified as dynamic by a division the more general it becomes.

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

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

As already mentioned, a binding-time analysis will, given a program $P$ (and some description on 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).

We will come back in the following subsection 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 $\Delta_0$ for $P$, a couple $(\mathcal{A}, \Delta)$ consisting of an annotation $\mathcal{A}$ and a division $\Delta$ for $P$ more general than $\Delta_0$. We will call the result of a binding-time analysis a *binding-time classification* ($BTC$)

The initial division $\Delta_0$ gives information about how the program will be specialised. In fact $\Delta_0$ specifies what the initial atom(s) to be specialised (i.e. the ones in $S_0$ of algorithm 2.1) can look like (if $p'$ does not occur in $S_0$ we simply set $\Delta_0(p') = (s, \ldots, s)$ We will assume that for all initial atoms $p$ we have $\Delta_0(p) = \Delta(p)$. If this is not the case for some predicate $p/n$ then one can ensure this by renaming $p$ to $p'$ where $p' \notin Pred(P)$ and adding a new clause $p(\overline{X}) \leftarrow p'(\overline{X})$ to $P$. The role of $\Delta$ is to give information about what the atoms in algorithm 2.1 will look like at the global level. In that light, not all $BTA$ as specified above are correct and we now develop a safety criterion for a $BTA$ wrt a given program. Basically a $BTA$ is safe iff every atom that can potentially appear in one of the sets $S_i$ of Algorithm 2.1 (given the restrictions imposed by the annotation of the $BTA$) corresponds to the patterns described by $\Delta$. Note that if a predicate $p$ is always unfolded by the unfold rule used in Algorithm 2.1 then it is irrelevant what the value of $\Delta_p$ is.

For simplicity we will from now on impose that a static argument must be ground.[5] In particular this guarantees that the argument will be neither more nor less bound at run-time.

**Definition 9.** (*safe wrt* $\Delta$)
Let $P$ be a program and let $\Delta$ 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* $\Delta$ iff $\Delta_p^s(\bar{t})$ is a tuple of ground terms. Also a goal $G$ is *safe wrt* $\Delta$ iff all the atoms occurring in $G$ are safe wrt $\Delta$.

**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.

[5] 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.

Then $\beta$ is a *safe BTC for $P$ and $U$* iff for every every goal $G$ which is safe wrt $\Delta$ it returns an incomplete SLDNF-tree whose leaf goals are safe wrt $\Delta$. Also $\beta$ 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. So, for the safety condition to capture safety at the global level, the abstraction operation used in algorithm 2.1 should not abstract atoms which are safe wrt $\Delta$ into atoms which are no longer safe wrt $\Delta$. One such abstraction operation will be defined in the next section.

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

### 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 App. A).

Let us first define a particular unfolding rule.

**Definition 11.** ($U_{\mathcal{L}}$)
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 2.1 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 $BTA$ to perform a pre-processing phase splitting the predicates according to the different uses.

The only thing that is missing in order to arrive at a concrete instance of algorithm 2.1 is the abstraction operation.

**Definition 12.** ($gen_\Delta$, $abstract_\Delta$)
Let $P$ be a program and $\Delta$ 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 $\Delta_p$) by distinct variables not occurring in A.
We also define the abstraction operation $abstract_\Delta$ to be the natural extension of the function $gen_\Delta$.

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 $gen_\Delta(q(a,b,c)) = q(X,b,c)$.

Note that $abstract_\Delta$ is a homomorphism and hence we can use a depth-first progression in algorithm 2.1 and still get the same specialisation. This is something which we will actually do in the practical implementation.

**Algorithm 2.2** *In the remainder of this paper we will use an off-line partial deduction method which does the following:*

1. *performs a BTA returning results of the form* $(\{U_\mathcal{L}\}, \Delta)$
2. *performs algorithm 2.1 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* $\Delta$.

**Proposition 13.** *Let* $(\{U_\mathcal{L}\}, \Delta)$ *be a safe BTC for a program* $P$. *Let* $S_0$ *be a set of atoms safe wrt* $\Delta$. *If algorithm 2.2 terminates then the final set* $S_i$ *only contains atoms safe wrt* $\Delta$.

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

*Example 2.* 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 $\Delta_0 = \{nont/3 \mapsto (s,d,d),\ t/3 \mapsto (s,s,s)\}$ 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 $\beta$ 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 $\Delta_0$ (and hence also wrt $\Delta$). 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) $nont(\mathrm{X},\mathrm{T},\mathrm{R}) \leftarrow t(a,\mathrm{T},\mathrm{V}), nont(\mathrm{X},\mathrm{V},\mathrm{R})$
(2) $nont(\mathrm{X},\mathrm{T},\mathrm{R}) \leftarrow t(\mathrm{X},\mathrm{T},\mathrm{R})$
(3) $t(\mathrm{X},[\mathrm{X}|\mathrm{Es}],\mathrm{Es}) \leftarrow$

**Fig. 1.** A parser

We conclude this section with some remarks on the relation between groundness analysis and $BTA$. As we restricted ourselves to *static* referring to ground

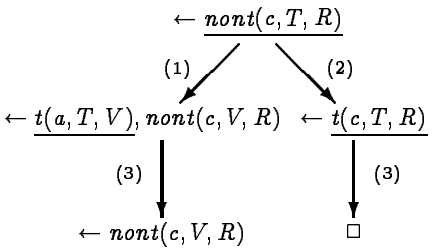


**Fig. 2.** Unfolding the parser of Fig. 1

terms one might think that the $BTA$ corresponds exactly to groundness analysis (via abstract interpretation [11] for instance). This is however not entirely true because a standard groundness analysis gives information about the arguments at the point where a call is 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 control level. So what we actually need is a groundness analysis adapted for unfolding rules and not for standard execution of logic programs. Actually the generating extensions we present in the next section contain a part performing the unfolding. In initial experiments we were able to successfuly extract BTCs by running the PLAI system (originally developed by [19]) with the set sharing domain [23] on this program part. Formalising and extending the approach is subject of ongoing research.

## 3 The cogen approach for logic programming languages

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 2.2) of any non-reducible atom $A$ which is safe wrt $\Delta$. So in the case of the parser from Ex. 2 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 $\beta$.

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 non-reducible predicates. Let us first consider the unfolding aspect of specialisation. The partial deduction algorithm unfolds the initial call to $p$ until there are no more reducible atoms to select 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 been unfolded previously. So all predicates may potentially be unfolded and the generating extension will, for each predicate $p/n$, contain a specific predicate $p_u$. 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. Doing this corresponds almost to executing the goal $p(t_1, ..., t_n)$ using the original predicate definition of $p$. To conform to $U_{\mathcal{L}}$ we only have to provide a way to prevent unfolding of the non-reducible predicates and a way to collect the atoms in the leaves.

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:

**Definition 14.** Let $P$ be a program and $C = p(\overline{t}) \leftarrow A_1, ..., A_k$ 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:

$$p_u(\overline{t}, [V_1, ..., V_k]) \leftarrow A_1^{V_1}, ..., A_k^{V_k}$$

where $V_1, ..., V_k$ are fresh variables not occurring in $\overline{t}, A_1, ..., A_k$ and where

1. $q(\overline{s})^V = q_u(\overline{s}, V)$, if $q \in \mathcal{L}$
2. $q(\overline{s})^V = eq(q(\overline{s}), V)$, if $q \notin \mathcal{L}$

and $eq$ is a predicate defined by just one fact $eq(X, X) \leftarrow$. We will denote by $P_u^{\mathcal{L}}$ the program obtained by applying the above transformation to every clause in $P$ and adding the definition of $eq$.

In the above definition calling $q_u(\overline{s}, V)$ corresponds to further unfolding whereas calling $eq(q(\overline{s}), V)$ corresponds to stopping the unfolding process. In the case of Ex. 2 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,V2]) :- t_u(a,T,V,V1),eq(nont(X,V,R),V2).
nont_u(X,T,R,[V1]) :- t_u(X,T,R,V1).
t_u(X,[X|R],R,[]).
eq(X,X).
```

Calling the above code with `nont_u(c,T,R)` 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)]
```

[6] For implementation reasons in unflattened form.

```
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 using Prolog's `findall` predicate. If one calls `findall(B,nont_u(c,R,T,B),Bs)` then `Bs` will be instantiated to `[[[],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),Bs)
```

we will even get in `Bs` a representation of the two resultants of Ex. 2.

Once all the resultants have been generated, the body atoms have to be generalised (using $gen_\Delta$) 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 of specialisation. That is, for every atom $p(\overline{t})$, if one calls $p_m(\overline{t}, V)$ then $V$ will be instantiated to the residual call of $p(\overline{t})$, but 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 $p \in \mathcal{L}$ we define the clause $C_m$ for the predicate $p_m$ to be:

$$
\begin{array}{l}
p_m(\overline{t}, V) \leftarrow \\
\quad (\mathit{find\_pattern}(p(\overline{t}), V) \rightarrow \mathit{true} \\
\quad\ ; \ (\mathit{insert\_pattern}(p(\overline{s}), H), \\
\qquad \mathit{findall}(C, (p_u(\overline{s}, B), \mathit{treat\_clause}(H, B, C)), Cs), \\
\qquad \mathit{pp}(Cs), \\
\qquad \mathit{find\_pattern}(p(\overline{t}), V)).
\end{array}
$$

where $p(\overline{s}) = gen_\Delta(p(\overline{t}))$. We will denote by $P_m^{\mathcal{L}}$ the program which is the union of all the $C_m$ clauses of $P$.

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 calls. So if the call to *find_pattern* succeeds, then $V$ has been instantiated to the residual call of $p(\overline{t})$, 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*$(C, (p_u(\overline{s}, B)$,*treat_clause*$(H, B, C)), Cs)$ unfolds the generalised atom $p(\overline{s})$ and returns a list of residual clauses for $p(\overline{s})$ (in $Cs$). The call to $p_u(\overline{s}, B)$ inside *findall* returns a leaf goal of the SLDNF-tree for $p(\overline{s})$. 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 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 $p(\overline{t})$ then $p_m(\overline{t}, V)$ is called and the atom is replaced by the value of $V$. 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 and the last call to *find_pattern* will instantiate $V$ to the residual call of the atom $p(\overline{t})$.

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. 2 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,F]) :- t_u(a,C,G,E), F = memo(nont(B,G,D)).
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 $p(\overline{t})$, 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 $p_m(\overline{t}, \_)$.

The job of the cogen is then quite simple: given a program $P$ and a safe *BTC* $\beta$ 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 for instance built-ins can be found in App. A.

## 4 Examples and Results

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

The first program is the parser from Ex. 2. 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 [31] in which the goals are "lifted" to the non-ground representation for resolution. This idea was first used by Gallagher in [13, 14]. A similar technique was put to good use in the self-applicable partial evaluator Logimix by Mogensen and Bondorf [38, 24]. Hill and Gallagher [20] also provide a recent account of this style of writing meta-interpreters with its uses and limitations. 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 [38]. In the experiment we conduct we use as binding-time annotation $dgenerate \mapsto (s, d)$, i.e. the regular expression is fully known whereas the string is dynamic.

The Tables 1, 2 and 3 summarise our benchmarks. The timings were obtained by using the *cputime*/1 predicate of Prolog by BIM on a Sparc Classic under Solaris (timings, at least for Table 1, were almost identical for a Sun 4).

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 as well and the speedups very satisfactory. The specialisation for the *parser* example corresponds to the one obtained in Ex. 2. 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 [13]. 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).
```

| 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 | $solve(\text{"}\{q(X) \leftarrow p(X), p(a) \leftarrow\}\text{"}, Q)$ |
| *regexp* | 0.03 s | $dgenerate(\text{"}(a + b) * .a.a.b\text{"}, S)$ |

**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 | $solve(\text{"}\{q(X) \leftarrow p(X), p(a) \leftarrow\}\text{"}, \text{"} \leftarrow q(a)\text{"})$ |
| *regexp* | 101.1 | $dgenerate(\text{"}(a + b) * .a.a.b\text{"}, \text{"}abaaaabbaab\text{"})$ |

**Table 3.** Running the specialised program

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 [38]. For further details see Appendix C

We also performed the experiments using some other specialisation systems. We tried out the self-applicable partial deducer SAGE (see [18]) for the logic programming language Gödel. We experimented also with the SP system (see [13]) for a subset of Prolog (comparable to our subset, with the exception that SP does not handle the *if-then-else*). Finally we did some experiments with the Paddy system (see [40]) written for full Eclipse (a variant of Prolog). We plan to do experiments with the Mixtus system ([42]) and with the self-applicable partial evaluator Logimix ([38]), but were held up due to technical difficulties in using these systems.

All systems were able to satisfactorily handle the *parser* example and came up with (almost) exactly the same specialised program as *cogen*.

The *solve* example however posed a bit more problems. Paddy basically came up with the same specialisation as *cogen* but left some useless tests and clauses inside. SP came up with the same specialisation as *cogen*, but only after re-specialising the specialised program a second time (also SP does not perform filtering which might account for some loss in efficiency). Finally SAGE performed little specialisation, returning almost the unspecialised program back.

Due to the heavy usage of the *if-then-else* the *regexp* example could only be tried with Paddy, which was able to specialise the program, but not to the extent of generating a deterministic automaton.

The only system which gave us access to the transformation time was Paddy. Note that the system runs under Eclipse and have for technical reasons to be executed on a Sun 4. The transformation times (to be compared with the results of Table 2) were $0.05s$ for the *parser* example, $0.8s$ for the *solve* example and $3.17s$ for the *regexp* example.

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 (a comparison here with the Logimix system would be very interesting as well and is planned in future experiments). Taking 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 one treated in this section) in [18] at least 7.9 hours (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 $447s$.

## 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. To make this more clear: 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 [13]) 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 programs, 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. One could say that our system allows divisions that are not uniformly congruent [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 and language paradigms [41, 21, 22, 5, 1, 17].

### Future Work

The most obvious goal of the near future is to see if a binding-time analysis can be developed, e.g. by extending or modifying an existing groundness/sharing analysis. On 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 (maybe even unfold/fold transformations [39] like tupling, goal replacement, etc.) can be incorporated into the cogen.

## Acknowledgements

We thank Danny De Schreye, André De Waal, Robert Glück, Gerda Janssens and Bern Martens 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 some valuable 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 enthusiasm.

## 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. To Appear.
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. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
12. 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.
13. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
14. 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.
15. 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.
16. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
17. R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, Lecture Notes in Computer Science, page to appear. Springer-Verlag, 1995.
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. D. Jacobs and A. Langen. Static analysis of logic programs for independent AND-parallelism. *Journal of Logic Programming*, 13(2 &3):291–314, May/July 1992.
24. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
25. 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.

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*, Lecture Notes in Computer Science 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.
29. M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In Logic Program Synthesis and Transformation. *Pre-Proceedings of LOPSTR'95*, Utrecht, Netherlands, September 1995. Extended version as Technical Report CW 216, K.U. Leuven.
30. M. Leuschel and D. De Schreye. An almost perfect abstraction operation for partial deduction using characteristic trees. Technical Report CW 215, Departement Computerwetenschappen, K.U. Leuven, Belgium, October 1995. Submitted for Publication.
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 and B. Martens. Global control for partial deduction through characteristic trees and global trees. Technical report, Departement Computerwetenschappen, K.U.Leuven, Belgium, December 1995. Submitted for Publication.
33. 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*, Portland, USA, December 1995. MIT Press. To appear. Extended version as Technical Report CW 210, K.U. Leuven.
34. J. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.
35. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
36. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 1995. To Appear.
37. 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. Extended version as Technical Report CSTR-94-16, University of Bristol.
38. 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.
39. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19 & 20:261–320, May 1994.
40. S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.
41. 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.
42. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

43. 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*, Portland, USA, December 1995. MIT Press. To appear.
44. 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. logical built-ins (=/2, not/1,...), extra logical built-ins (=../2, call/1,...) or externally defined (by a 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:

1. $q(\overline{s})^V = q(\overline{s}), eq(\texttt{[]}, V)$, if $q(\overline{s})$ is marked reducible
2. $q(\overline{s})^V = eq(q(\overline{s}), V)$, if $q(\overline{s})$ is marked non-reducible

As a last example of how to extend the method we will show how to handle the Prolog version of the conditional: `(If → Then ; Else)`.

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.

$$(G_1 \ \rightarrow \ G_2 \ ; \ G_3)^V = (G_1 \ \rightarrow \ G_2^V \ ; \ G_3^V)$$

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. So if $A_1$, $A_2$ and $A_3$ are goals that satisfy the restriction above the generating extension will simply be

$$(A_1 \underline{\rightarrow} A_2 \ \underline{;} \ A_3)^V = A_1^{V_1}, A_2^{V_2}, A_3^{V_3}, eq(V, (V_1 \ \rightarrow \ V_2 \ ; \ V_3))$$

The restriction on the subgoals ensures that the three goals $\{A_i^{V_i} | i = 1, 2, 3\}$ 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

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

/* the file .ann contains:
   ann_clause(Head,Body),
   delta(Call,StaticVars,DynamicVars),
   residual(P) */

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],[G1|GS1],[V|VS]) :-
  body(G,G1,V),
  bodys(GS,GS1,VS).

body(unfold(Call),ResCall,V) :-
  add_extra_argument("_u",Call,V,ResCall).
body(memo(Call),V=memo(Call),V).
body(call(Call),Call,[]).
body(rescall(Call),V=rescall(Call),V).
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,(V=if(VS1,VS2,VS3))],V) :-
  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.
```

## C The Regular Expression Example

The annotated program looks like:

```
static_consult(['regexp.calls']).
delta(dgenerate(RX,S),[RX],[S]).
residual(dgenerate(_,_)).

ann_clause(dgenerate(RegExp,[]),
        [call(nullable(RegExp))]).

ann_clause(dgenerate(RegExp,[C|T]),
        [call(first(RegExp,C2)),
         call(dnext(RegExp,C2,NextRegExp)),
         call(C2=C),
         memo(dgenerate(NextRegExp,T))]).
```

The `static_consult` primitive tells *cogen* that some auxiliary predicates are defined in the file `regexp.calls`. This will translate to a `consult` being inserted

into the generating extension. The file `regexp.calls` contains the definitions of `first`, `dnext` and `nullable`.

The generating extension produced by *cogen* for the annotation *dgenerate*$(s, d)$:

```
/*  --------------------  */
/*  GENERATING EXTENSION  */
/*  --------------------  */
:- reconsult(memo).
:- reconsult(pp).
:- consult('regexp.calls').

dgenerate_m(B,C,D) :-
  ((
    find_pattern(dgenerate(B,C),D)
   ) -> (
    true
   ) ; (
    insert_pattern(dgenerate(B,E),F),
    findall(G, (
      ','(dgenerate_u(B,E,H),treat_clause(F,H,G))),I),
    pp(I),
    find_pattern(dgenerate(B,C),D)
  )).
dgenerate_u(B,[],[[]]) :-
  nullable(B).
dgenerate_u(C,[D|E],[[],[],[],F]) :-
  first(C,G),
  dnext(C,G,H),
  (G) = (D),
  (F) = (memo(dgenerate(H,E))).
```

Running the generating extension for

```
dgenerate(cat(star(or(a,b)),cat(a,cat(a,b))),String)
```

yields the following program corresponding to a deterministic automaton for the regular expression $(a + b)^*aab$:

```
dgenerate__3([]).
dgenerate__3([a|B]) :- dgenerate__1(B).
dgenerate__3([b|C]) :- dgenerate__0(C).

dgenerate__2([a|B]) :- dgenerate__2(B).
dgenerate__2([b|C]) :- dgenerate__3(C).

dgenerate__1([a|B]) :- dgenerate__2(B).
dgenerate__1([b|C]) :- dgenerate__0(C).

dgenerate__0([a|B]) :- dgenerate__1(B).
dgenerate__0([b|C]) :- dgenerate__0(C).
```

This article was processed using the LaTeX macro package with LLNCS style