Efficient Specialisation in Prolog Using a Hand-Written Compiler Generator

Michael Leuschel and Jesper Jørgensen
mal@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-99-6
September 15 1999

www.dsse.ecs.soton.ac.uk/techreports/

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Efficient Specialisation in Prolog Using a Hand-Written Compiler Generator

Michael Leuschel* and Jesper Jørgensen†

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 logic programming as for 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 should prove to have a considerable importance when it comes to practical applications.

This paper first develops a generic offline partial deduction technique for pure logic programs, notably supporting partially instantiated datastructures via binding types. From this a very efficient *cogen* is derived, which generates very efficient generating extensions (executing up to several orders of magnitude faster than current online systems) which in turn perform very good and non-trivial specialisation, even rivalling existing online systems. All this is supported by extensive benchmarks. Finally, it is shown how the *cogen* can be extended to directly support a large part of Prolog's declarative and non-declarative features and how semi-online specialisation can be efficiently integrated.

1 Introduction

Partial evaluation has over the past decade received considerable attention both in functional (e.g. [31]), imperative (e.g. [1]) and logic programming (e.g. [19, 35, 60]). In the context of pure logic programs, partial evaluation is sometimes

*Declarative Systems and Software Engineering, Dept. of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, UK, TEL +44 2380 59 3377, FAX: +44 2380 59 3045, E-MAIL: mal@ecs.soton.ac.uk, WWW: http://www.ecs.soton.ac.uk/~mal

†Dept. of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark, E-MAIL:jesper@dina.kvl.dk.

referred to as *partial deduction*, the term partial evaluation being reserved for the treatment of impure logic programs.

Guided by the *Futamura projections* (see e.g. [31]) 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).

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 [64], or PADDY [62]) has been made effectively self-applicable. On the other hand a partial deducer which specialises only purely declarative logic programs (like SAGE in [26] 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 [17] and [57]. However the specialisation in [17] 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 [57, 38]). The partial evaluator LOGIMIX [57] does not share this problem, but gives only modest speedups (when compared to results for functional programming languages, see the remarks in [57]) when self-applied.

However, the actual creation of the *cogen* according to the third Futamura projection is not of much interest to users since *cogen* can be generated once and for all when a specialiser is given. Therefore, from a user's 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 is efficient and produces efficient, non-trivial compilers. 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 the *cogen* turns out to be just a simple extension of a "binding-time analysis" for logic programs (something first discovered for functional languages in [29]).

In this paper we will describe the first *cogen* written in this way for a logic programming language. We start out by a *cogen* for a small subset of Prolog

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

and progressively improve it to handle a large part of Prolog and extend its capabilities.

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 do not have to be self-applied)! As we will see, this leads to extremely efficient compilers and compiler generators.

Some general advantages of the *cogen* approach are: the *cogen* manipulates only syntax trees and there is no need to implement a self-interpreter[2]; 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. This is in contrast to self-applicable partial deducers which *must* use the ground representation in order to be declarative (see [28, 47, 26]). 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 [47, 43], can be impossible to specialise satisfactorily by partial deduction alone. ([57] uses a kind of "mixed" representation where programs are ground and goals non-ground; see also [42, 28]).

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 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:

1. a formal specification of the concept of *binding-time analysis* and more generally *binding-type analysis*, allowing the treatment of *partially static* structures, in a (pure) logic programming setting and a description of how to obtain a generic algorithm for *offline partial deduction* from such an analysis.
2. based upon point 1, the first description of an efficient, handwritten compiler generator (*cogen*) which shows that such a program has a quite elegant and natural structure.
3. a way to handle both *extra-logical* features (such as `var/1` or the if-then-else) and *side-effects* (such as `print/1`) within the *cogen*. A refined treat-

[2] I.e. a meta-interpreter for the underlying language. Indeed the *cogen* just transforms the program to be specialised, yielding a compiler which is then evaluated by the underlying system (and not by a self-interpreter).

ment of the `call/1` predicate is also presented.

4. how to handle negation, disjunction and the if-then-else conditional in the *cogen*.
5. extensive benchmark results showing the efficiency of the *cogen*, the generating extensions but also of the specialised programs.

This paper is a much extended and revised version of [33]: points 3, 4, 5 and the partially static structures of point 1 are new, leading to a much more powerful, practical and viable *cogen*.

The paper is organised as follows: In Section 2 we formalise the concept of off-line partial deduction and the associated binding-type analysis. In Section 3 we present and explain our *cogen* approach in a pure logic programming setting, starting from the structure of the generating extensions. The issue of built-in's is discussed in Section 4. We discuss the treatment of declarative and non-declarative built-in's but also constructs such as negations, conditionals, and disjunctions. In Section 5 we present benchmarks underlining the efficiency of the *cogen* and of the generating extensions it produces. We conclude with some discussions of related and future work in Section 6.

2 Off-line Partial Deduction

Throughout this paper, we suppose familiarity with basic notions in logic programming ([51]). 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.

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 finite but possibly incomplete[3] SLDNF-trees. The derivation steps in these SLDNF-trees correspond to the computation steps which have already been performed by the *partial deducer* and the clauses of the specialised program are then extracted from these trees by constructing one specialised clause (called a *resultant*) per branch. These incomplete SLDNF-trees are obtained by applying an unfolding rule, defined as follows.

[3] 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.

Definition 1. An *unfolding rule* is a function which, given a program P and a goal G, returns a non-trivial[4] and possibly incomplete SLDNF-tree for $P \cup \{G\}$.

The following formally defines resultants.

Definition 2. Let P be a normal program and A an atom. Let τ be a finite, incomplete SLDNF-tree for $P \cup \{\leftarrow A\}$. Let $\leftarrow G_1, \ldots, \leftarrow G_n$ be the goals in the 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 uses the resultants for a given set of atoms $\mathcal{S}$ to construct the specialised program (and for each atom in $\mathcal{S}$ a different specialised predicate definition will be generated). Under the conditions stated in [52], namely *closedness* (all leaves are an instance of an atom in $\mathcal{S}$) and *independence* (no two atoms in $\mathcal{S}$ have a common instance), correctness of the specialised program is guaranteed.

In a lot of practical approaches (e.g. [18, 19, 21, 42, 45, 39, 49]) independence is ensured by using a *renaming* transformation which maps dependent atoms to new predicate symbols. Adapted correctness results can be found in [4] [49] and [46]. Renaming is often combined with argument *filtering* to improve the efficiency of the specialised program (see e.g. [20, 4] and also [50]).

Closedness can be ensured by using the following outline of a partial deduction algorithm (similar to the ones used in e.g. [18, 19, 39, 44]).[5]

Algorithm 3. (Partial deduction)

```
Input: a program $P$ and an initial set $\mathcal{S}_0$ of atoms to be specialised
Output: a set of atoms $\mathcal{S}$
Initialisation: $\mathcal{S}_{new} := abstract(\mathcal{S}_0)$
repeat
    $\mathcal{S}_{old} := \mathcal{S}_{new}$
    $S_{new} := \{s_n \mid s_n \in leaves(U_P(s_o)) \wedge s_o \in \mathcal{S}_{old}\}$
    $S_{new} := abstract(\mathcal{S}_{old} \cup S_{new})$
until $\mathcal{S}_{old} = \mathcal{S}_{new}$ (modulo variable renaming)
output $\mathcal{S} := \mathcal{S}_{new}$
```

[4] A trivial SLDNF-tree is one in which no literal in the root has been selected for resolution. Such trees are disallowed to obtain correct partial deductions (c.f., Definition 2).

[5] More recent (online) algorithms structure the atoms to be specialised in a tree rather than in a set [55, 49].

The above algorithm is parametrised by an unfolding rule U_P and an abstraction operation *abstract*. The latter is used to ensure termination and can be formally defined as follows.

Definition 4. An *abstraction operation* is a function *abstract* from sets of atoms to sets of atoms such that for any finite set of atoms S:

> *abstract*(S) is a finite set of atoms S' with the same predicates as those in S, and every atom in S is an instance of an atom in S'.

If the Algorithm 3 terminates then the closedness condition is satisfied.

2.2 Off-Line Partial Deduction and Binding-Types

In Algorithm 3 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* [19, 55]. The abstraction operation controls the construction of the set of atoms for which such 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. 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 be required. This analysis phase provides annotations which then guide the specialisation phase proper, often to the point of making it almost trivial.

Partial evaluation of functional programs ([13, 31]) has mainly stressed off-line approaches, while supercompilation of functional ([70, 67]) and partial deduction of logic programs ([21, 64, 6, 9, 54, 55, 39, 48, 49, 34]) have concentrated on on-line control. (Some exceptions are [57, 42, 38].)

The main reason for using the off-line approach is to achieve effective self-application ([32]). But the off-line approach is in general also much 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. The purpose of this analysis is to figure out which values will be known at specialisation time proper and which values will only be known at runtime. The simplest approach is to classify arguments within the program to be specialised as either *static* or *dynamic* (see Figure 1). The value of a static argument will be *definitely known* (bound) at specialisation time whereas a dynamic argument is not necessarily known at specialisation time. In the context of partial deduction of logic programs, a static argument can be seen [57] as being a term which is guaranteed not to be more instantiated at run-time (it can never be less instantiated at run-time; otherwise the information provided would be incorrect). 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[6] — actual run-time instances might be $p(a, b), p(a, Z), p(a, X)$ but not $p(b, c)$.

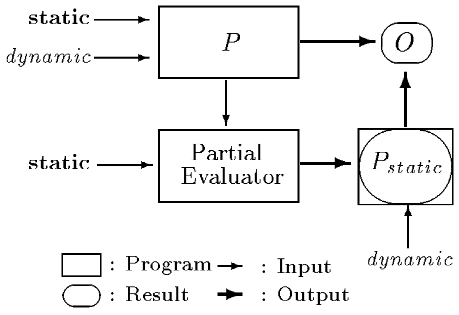

Figure 1: Partial evaluation of programs with static and dynamic input

This approach is sufficient for functional programs, but often proves to be too weak for logic programs: in logic programming partially instantiated datastructures appear naturally even at runtime. A simple classification of arguments into "fully known" or "totally unknown" is therefore unsatisfactory and would prevent specialising a lot of "natural" logic programs such as the vanilla metainterpreter [28, 53] or most of the benchmarks from the DPPD library [40].

The basic idea to improve upon the above shortcoming, is to describe which parts of arguments will actually be known at specialisation time by a special form of types.[7] Below, we will develop the first such description, of what we call *binding-types*, in logic programming.

Binding-Types

In logic programming, a type is just a decidable set of terms closed under substitution [2]. We will adapt the definitions and concepts of [72], which mainly follow the Hilog notation [11].

Types are built up from type variables and type constructors in much the same way as terms are built-up from ordinary variables and function symbols. Formally, a *type* is either a *type variable* or a *type constructor* of arity $n \geq 0$ applied to n types. We pre-suppose the existence of three unary type constructors:

[6] However, if X is actually an existential variable then one might also consider it static.

[7] This is somewhat related to the way instantiations are defined in the Mercury language [65]. But there are major differences, which we discuss later.

`static`, `dynamic`, and `nonvar`. These constructors will be given a pre-defined meaning below.

Definition 5. A *type definition* for a type constructor c of arity n is of the form

$$c(V_1,\ldots,V_n) ::= f_1(T_1^1,\ldots,T_1^{n_1}) \ ; \ \ldots \ ; \ f_k(T_k^1,\ldots,T_k^{n_k}) \quad (n, n_j \geq 0, k \geq 1)$$

where f_i is a function symbol, $V_1,\ldots,V_n$ are distinct type variables and T_i^j are types which only contains type variables in $\{V_1,\ldots,V_n\}$.
A *type system* Γ is set of type definitions, exactly one for every type constructor c different from `static`, `dynamic`, and `nonvar`. We will refer to the type definition for c in Γ by $Def_{\Gamma}(c)$.

From now on we will suppose that the underlying type system Γ is fixed. A possible type system Γ_1 is as follows:

```
:- type list(T) --> [] ; [T | list(T)].
:- type listskel ---> list(dynamic).
```

We define *type substitutions* to be finite sets of the form $\{V_1/\tau_1,\ldots,V_k/\tau_k\}$, where every V_i is a type variable and τ_i a type. Type substitutions can be applied to types (and type definitions) to produce *instances* in exactly the same way as substitutions can be applied to terms. For example, $list(V)\{V/static\} = list(static)$. A type or type definition is called *ground* if it contains no type variables.

Definition 6. We now define *type judgements* relating terms to types in the type system Γ.

- $t : dynamic$ holds for any term t
- $t : static$ holds for any ground term t
- $t : nonvar$ holds for any non-variable term t
- $f(t_1,\ldots,t_n) : c(\tau_1',\ldots,\tau_k')$ if $t_i : \tau_i$ for $1 \leq i \leq n$ and if there exists a ground instance of the type definition $Def_{\Gamma}(c)$ which has the form $c(\tau_1',\ldots,\tau_k') ::= \ldots f(\tau_1,\ldots,\tau_n)\ldots$

We also say that a type τ is *more general* than another type τ' iff whenever $t : \tau'$ then also $t : \tau$.

For example, using the type system Γ_1 above we have amongst others $s(0) : static$, $s(0) : dynamic$, $s(X) : dynamic$, $X : dynamic$, $[] : list(static)$, $[s(0)] : list(static)$, $[X,Y] : listskel$. We also have that $list(dynamic)$ is more general than $listskel$. Also note that types are downwards-closed (i.e., $t : \tau \Rightarrow t\theta : \tau$).

Binding-Type Analysis and Classification

We will now formalise the concept of a *binding-type* analysis (which is an extension of a binding-*time* analysis, as in [33]). For that we first define the concept of a division which assigns types to arguments of predicates.

Definition 7. A *division for a predicate* p of arity n is an expression of the form $p(\tau_1, \ldots, \tau_n)$ where each τ_i is a ground type.
A *division for a program* P is a set of divisions for predicates in $Pred(P)$. When there is no ambiguity about the underlying program P we will also often simply refer to a *division*.
A division is called *simple* iff it contains only the types `static` and `dynamic`.
A division is called *monovariant* iff it contains at most one division for any predicate p. A division Δ is called *more general* than another division Δ' iff for every $p(\tau'_1, \ldots, \tau'_n) \in \Delta'$ there exists $p(\tau_1, \ldots, \tau_n) \in \Delta$ such that for $1 \leq i \leq n$ τ_i is more general than τ'_i.

Now, a *binding-type 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 every predicate in P describing the part of the values that will be known at specialisation time. It will also return an *annotation* which will then guide the local unfolding process of the actual partial deduction. For the time being, an annotation can simply be seen as a particular unfolding rule $\mathcal{U}$. To be called off-line, $\mathcal{U}$ should as simple-minded as possible and, e.g., not take the unfolding history into account. We will return to this in Section 2.3.

We are now in a position to formally define a binding-type analysis in the context of (pure) logic programs:

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

The purpose of the initial division Δ_0 is to give information about how the program will be specialised: it specifies what form the initial atom(s) (i.e., the ones in $\mathcal{S}$ of Algorithm 3) can take. The rôle of Δ is to give information about the possible atoms that can occur at the global level (i.e., the ones in $\mathcal{S}_{new}$ and $\mathcal{S}_{old}$ of Algorithm 3). In that light, not all *BTC* are correct and we have to develop a safety criterion. Basically a *BTC* is safe iff every atom that can potentially appear in one of the sets $\mathcal{S}_{new}$ and $\mathcal{S}_{old}$ of Algorithm 3 (given the restrictions imposed by the annotation of the *BTA*) corresponds to the patterns described by Δ.

Definition 9. (*safe wrt* Δ)
Let P be a program and let Δ be a division for P and let $p(t_1, \ldots, t_n)$ be

an atom. Then $p(t_1, \ldots, t_n)$ is *safe wrt* Δ iff $p(\tau_1, \ldots, \tau_n) \in \Delta$ implies that $\forall i \in \{1, \ldots, n\}$ we have $t_i : \tau_i$. 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 $p(a, X)$ and $p(a, a)$ are safe wrt $\Delta = \{p(static, dynamic)\}$ while $p(X, a)$ is not.

Definition 10. (safe BTC, safe BTA)
Let $\beta = (\mathcal{U}, \Delta)$ be a BTC for a program P. Then β is a *safe BTC for P* iff for every goal G which is safe wrt Δ, $\mathcal{U}$ returns an SLDNF-tree τ for $P \cup \{G\}$ whose leaf goals are safe wrt Δ. Furthermore, β is called *LD-safe for P* if in all trees τ all the atoms to the left of selected literals are safe wrt Δ.
A BTA is *safe* if for any program P it produces a safe BTC for P.

The reason for introducing LD-safety is that sometimes (for efficiency reasons) one might want to abstract and specialise atoms before the full SLDNF-tree τ has been built; i.e., not necessarily the leaves of τ will be added but (less instantiated) atoms within the tree. We will return to this issue in Section 3.3.

Notice, that in the above definition of safety no requirement is made about the actual atoms selected by $\mathcal{U}$. Indeed, contrary to functional or imperative programming languages, logic programming can handle uninstantiated variables and a positive atom can always be selected. Nonetheless, if we have negative literals or Prolog built-in's, this is no longer true. For example, `X is Y + 1` can only be selected if `Y` is ground. Put in other terms, we can only select a call "s `is` t" if it is safe wrt $\{is(dynamic, static)\}$. Also, we might want to restrict unfolding of user-defined predicates to cases where only one clause matches. For example, we might want to unfold a call $app(r, s, t)$ (see Example 1 below) only if it is safe wrt $\{app(static, dynamic, dynamic)\}$. This motivates the next definition, which can be used to ensure that only properly instantiated built-in's and atoms are selected.

Definition 11. A BTC $\beta = (\mathcal{U}, \Delta)$ is *locally safe for P* iff for every goal G which is safe wrt Δ, $\mathcal{U}$ produces an SLDNF-tree for $P \cup \{G\}$ whose selected literals are safe wrt Δ.

One can easily allow the use of different divisions for local and global safety (but we will not do so in the presentation).

Let us now return to the global control. Definition 10 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, in order for Definition 10 to ensure safety at all stages of Algorithm 3, the particular abstraction operation employed should not abstract atoms which are safe wrt Δ into atoms which are no longer safe wrt Δ. This motivates the following definition:

Definition 12. 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 .

In particular this means that *abstract* can only generalise positions marked as `dynamic` or the arguments of positions marked as `nonvar` within the respective binding-type. For example, $abstract(\{p([])\}) = \{p(X)\}$ is neither safe wrt $\Delta = \{p(static)\}$ nor wrt $\Delta' = \{p(listskel)\}$ but is safe wrt $\Delta'' = \{p(dynamic)\}$. Also, $abstract(\{p(f([]))\}) = \{p(f(X))\}$ is not safe wrt $\Delta = \{p(static)\}$ but is safe wrt $\Delta'' = \{p(nonvar)\}$.

Example 1. Let P be the well known append program

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

Let $\Delta = \{app(static, dynamic, dynamic)\}$ and let $\mathcal{U}$ be any unfolding rule. Then $(\mathcal{U}, \Delta)$ is a safe and locally safe *BTC* for P. For example the goal $\leftarrow app([a, b], Y, Z)$ is safe wrt Δ and an unfolding rule can either stop at $\leftarrow app([b], Y, Z)$, $\leftarrow app([], Y', Z')$ or at the empty goal $\Box$. All of these goals are safe wrt Δ. More generally, unfolding a goal $\leftarrow app(t_1, t_2, t_3)$ where t_1 is ground, leads only to goals whose first arguments are ground.

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, until further notice, 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 (both of which we will discuss later).

Let us first define a particular class of simple-minded but "efficient" unfolding rules.

Definition 13. ($U_{\mathcal{A}}$)
An *annotation* $\mathcal{A}$ for a program P marks every literal in the body of a clause of P as either *reducible* or *non-reducible.* A program P together with an annotation $\mathcal{A}$ for it is called an *annotated program*, denoted by $P_{\mathcal{A}}$. We then define the unfolding rule $U_{\mathcal{A}}$ to be the unfolding rule which always unfolds the atom in the root and then keeps track of which literals have been marked as reducible and selects the leftmost reducible atom in each goal.

Syntactically we represent an annotation for P by underlining the predicate symbol of reducible literals.[8]

[8] In functional programming one usually underlines the non-reducible calls. But in logic programming underlining a literal is usually used to denote selected literals and therefore underlining the reducible calls is more intuitive.

Example 2. Let $P_{\mathcal{A}}$ be the following annotated program

$$\begin{array}{l} p(X) \leftarrow \underline{q}(X,Y), \underline{q}(Y,Z) \\ q(a,b) \leftarrow \\ q(b,a) \leftarrow \end{array}$$

Let $\Delta = \{p(static), q(static, dynamic)\}$. Then $\beta = (U_{\mathcal{A}}, \Delta)$ is a safe BTC for P. For example the goal $\leftarrow p(a)$ is safe wrt Δ and unfolding it according to $U_{\mathcal{A}}$ 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 and β is actually also locally safe for P. Also note that $\beta' = (U_{\mathcal{A}'}, \Delta)$, where $\mathcal{A}'$ marks every literal as non-reducible, is a *not* a safe BTC for P. For instance, given the goal $\leftarrow p(a)$ the unfolding rule $U_{\mathcal{A}'}$ just performs one unfolding step and thus stops at the goal $\leftarrow q(a,Y), q(Y,Z)$ which contains the unsafe atom $q(Y,Z)$.

From now on we will only use unfolding rules of the form $U_{\mathcal{A}}$ obtained from an annotation $\mathcal{A}$ and our BTA's will thus return results of the form $\beta = (U_{\mathcal{A}}, \Delta)$. For reasons of simplicity of the presentation, will also restrict ourselves to monovariant BTA's. This is not really a limitation, as nothing prevents us from performing a pre-processing phase which splits predicates according to their different uses, allowing the same predicate to be given different binding-types and different annotations depending on the context.

In order to arrive at a concrete instance of Algorithm 3 we now only need a (safe) abstraction operation, which we define in the following.

Definition 14. (gen_Δ, $abstract_\Delta$)
We first define mappings gen_τ from atoms to atoms by the following rules:

- $gen_{static}(t) = t$, for any term t
- $gen_{dynamic}(t) = V$, for any term t and where V is a fresh variable
- $gen_{nonvar}(f(t_1, \ldots, t_n)) = f(V_1, \ldots, V_n)$, where $\{V_1, \ldots, V_n\}$ are n distinct fresh variables
- $gen_{c(\tau'_1, \ldots, \tau'_k)}(f(t_1, \ldots, t_n)) = f(u_1, \ldots, u_n)$, where $gen_{\tau_i}(t_i) = u_i$ and a ground instance in $Def_\Gamma(c)$ has the form $c(\tau'_1, \ldots, \tau'_k) ::= \ldots f(\tau_1, \ldots, \tau_n) \ldots$

Let P be a program and Δ be a monovariant division for P. Also let $p(\tau_1, \ldots, \tau_n) \in \Delta$. We then define $gen_\Delta(p(t_1, \ldots, t_n) = p(gen_{\tau_1}(t_1), \ldots, gen_{\tau_n}(t_n))$.
We also define the abstraction operation $abstract_\Delta$ as follows: $abstract_\Delta(S)$ is obtained from the set $\{gen_\Delta(s) \mid s \in S\}$ by removing any superfluous variants.

For example, if $\Delta = \{p(static, dynamic), q(dynamic, static, nonvar)\}$ then $gen_\Delta(p(a,b)) = p(a,X)$ and $gen_\Delta(q(a,b,f(c))) = q(X,b,f(Y))$. We also have that $abstract_\Delta(\{p(a,b), q(a,b,f(c))\}) = \{p(a,X), q(X,b,f(Y))\}$. Finally, for $\Delta' = \{r(listskel)\}$ (where $listskel$ is defined in Section 2.2) we have that $gen_\Delta(r([a,b,c])) = r([X,Y,Z])$ and $gen_\Delta(r([H|T]))$ is undefined.

Proposition 15. *For every division Δ, $abstract_\Delta$ is safe wrt Δ.*

Note that in Algorithm 3 the atoms in $leaves(U_P(s_o))$ 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). However, $abstract_\Delta$ is a homomorphism[9] (up to variable renaming) and thus both progressions will yield exactly the same set of atoms and thus the same specialisation. This is something which we will actually exploit in the practical implementation which uses the more efficient depth-first progression.

Based upon this abstraction operation, we can also define a corresponding renaming and filtering operation:

Definition 16. Let $||.||$ be a fixed mapping from atoms to natural numbers such that $||A|| = ||B||$ iff A and B are variants. We then define filter_Δ as follows: $\mathit{filter}_\Delta(A) = p_{||gen_\Delta(A)||}(V_1\theta, \ldots, V_k\theta)$, where $A = p(\bar{t})$ is the atom to be filtered, $p(\bar{t}) = gen_\Delta(A)\theta$, and $vars(gen_\Delta(A)) = \{V_1, \ldots, V_k\}$.

The purpose of the above is to assign every specialised atom (i.e., atoms of the form $gen_\Delta(A)$) a unique identifier id and predicate name, thus ensuring the independence condition [52]. The operation will properly rename instances of these atoms and also filter out static parts, thus improving the efficiency of the residual code [20, 4]. For example, given the division $\Delta = \{p(static, dynamic), q(dynamic, static, nonvar)\}$, $||p(a, X)|| = 1$, and $||q(X, b, f(Y)|| = 2$ we have $\mathit{filter}_\Delta(p(a, b)) = p_1(b)$ and $\mathit{filter}_\Delta(q(a, b, f(c))) = q_2(a, c)$.

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

Algorithm 17. (off-line partial deduction)

1. Perform a BTA (possibly by hand) returning results of the form $(U_\mathcal{A}, \Delta)$
2. Perform Algorithm 3 with $U_\mathcal{A}$ as unfolding rule and $abstract_\Delta$ as abstraction operation. The initial set of atoms $\mathcal{S}_0$ should only contain atoms which are safe wrt Δ.
3. Construct the specialised program using filter_Δ.

Proposition 18. *Let $(\{U_\mathcal{A}\}, \Delta)$ be a safe BTC for a program P. Let $\mathcal{S}$ be a set of atoms safe wrt Δ. Then all sets $\mathcal{S}_{new}$ and $\mathcal{S}_{old}$ arising during the execution of Algorithm 17 are safe wrt Δ.*

Notably, if Algorithm 17 terminates the final set $\mathcal{S}$ will be safe wrt Δ.

We now illustrate this particular partial deduction method on a relatively simple example.

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

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 [35] and the parser P is depicted in Figure 2.

Given the initial division $\Delta_0 = \{nont(static, dynamic, dynamic)\}$ a BTA might return the result $\beta = (U_{\mathcal{A}}, \Delta)$ where $\Delta = \{nont(static, dynamic, dynamic), t(static, dynamic, dynamic)\}$ and where $\mathcal{A}$ is represented in Fig. 2. It can be seen that β is a safe and locally safe BTC for P.

Let us now perform the proper partial deduction for $\mathcal{S}_0 = \{nont(c, R, T)\}$. Note that the atom $nont(c, R, T)$ is safe wrt Δ_0 (and hence also wrt Δ). Unfolding the atom in $\mathcal{S}_0$ yields the SLD-tree in Fig. 4. We see that the atoms in the leaves are $\{nont(c, V, T)\}$ and we obtain $\mathcal{S}_{old} = \mathcal{S}_{new}$ (modulo variable renamig). The specialised program after renaming and filtering is depicted in Figure 3.

$$\boxed{\begin{array}{ll} nont(X,T,R) \leftarrow & \underline{t}(a,T,V), nont(X,V,R) \\ nont(X,T,R) \leftarrow & \underline{t}(X,T,R) \\ t(X,[X|ES],ES) \leftarrow & \end{array}}$$

Figure 2: A very simple parser

$$\boxed{\begin{array}{l} nont_1([a|V],R) \leftarrow nont_1(V,R) \\ nont_1([c|R],R) \leftarrow \end{array}}$$

Figure 3: A specialisation of Figure 2

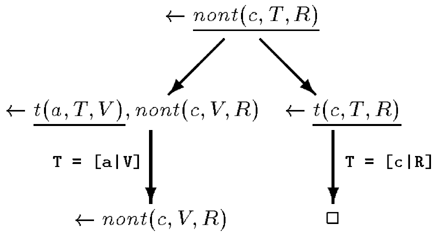
$\leftarrow \underline{nont(c,T,R)}$
$\leftarrow \underline{t(a,T,V)}, nont(c,V,R)$
$\leftarrow \underline{t(c,T,R)}$
T = [a|V]
T = [c|R]
$\leftarrow nont(c,V,R)$
□

Figure 4: Unfolding the parser of Figure 2

3 The *cogen* approach for logic programming

Based upon the generic offline partial deduction framework presented in the previous section, we will now describe the *cogen* approach to logic program specialisation.

3.1 General Overview

In that context of our framework, a *generating extension* for a program P wrt to a given safe BTC $(U_{\mathcal{A}}, \Delta)$ for P, is a program that specialises (using parts 2 and 3 of Algorithm 17) any atom S safe wrt Δ, thereby producing a specialised program P_S. In the particular context of Example 3 a generating extension is a program that, when given the safe atom $nont(c, R, T)$, produces the residual program shown in Figure 3.

A *compiler generator*, *cogen*, is a program that given a program P and a safe BTC $\beta = (U_{\mathcal{A}}, \Delta)$ for P, produces a generating extension for P wrt β.

An overview of the whole process is depicted in Figure 5 (the κ, γ, and σ subscripts will be explained in the next section). As can be seen, P, Δ, and $\mathcal{A}$ have been compiled into the generating extension $genex^P_{\mathcal{A},\Delta}$ (contributing to its efficiency and also making it standalone). A generating extension is thus not a generic partial evaluator, but a highly specialised one: it can just specialise the program P for safe calls S wrt Δ and it can only follow the annotation $\mathcal{A}$.

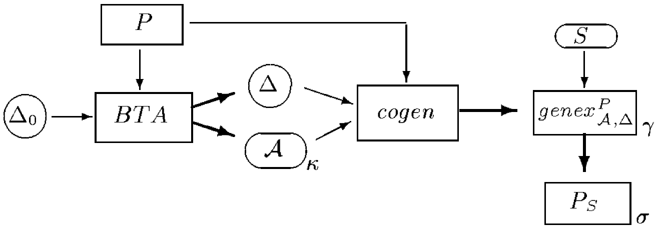

Figure 5: Overview of the *cogen* approach

To explain and formalise the *cogen* approach, we will first examine the rôle and structure of efficient generating extensions $genex^P_{\mathcal{A},\Delta}$. Once this is clear we will consider what *cogen* should look like.

3.2 The local control

Let us first consider the local control aspect. The crucial idea for simplicity and efficiency of the generating extensions is to incorporate a specific "unfolding" 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 will collect 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 $U_{\mathcal{A}}$ to $p(t_1, ..., t_n)$, whereby it will return in B the atoms in the leaf of the branch[10] and also instantiate $t_1, ..., t_n$ via the composition of *mgu*'s of the branch (see Figure 6). 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 of non-reducible atoms can be prevented and the corresponding leaf atoms be collected in the last argument B.

$$\begin{tikzcd}
p(t_1, ..., t_n) \arrow[d, "U_{\mathcal{A}}:\ \theta"'] & \rightsquigarrow_u & p_u(t_1, ..., t_n, B) \arrow[d, "\theta + B/[L_1, ..., L_m]"] \\
L_1, ..., L_m & & \square
\end{tikzcd}$$

Figure 6: Going from p to p_u

All this can be obtained very easily by transforming every clause for p/n into a clause for $p_u/(n+1)$ in the following manner.

Definition 19. We first define the ternary relation $\kappa \rightsquigarrow \gamma : \sigma$. Intuitively (see also Figure 5), $\kappa \rightsquigarrow \gamma : \sigma$ denotes that the *cogen* will produce from the annotated literal or conjunction κ the code γ within the generating extension. In turn, executing γ will instantiate the free variables in σ so that it represents the specialised code proper. If γ fails then no residual clause will be produced. On the other hand if γ has several computed answers then several residual clauses will be produced.

The following 3 rules (in natural deduction style) define the relation:

$\underline{p}(\overline{t}) \rightsquigarrow p_u(\overline{t}, C) : C$ (C fresh variable)

$p(\overline{t}) \rightsquigarrow p_m(\overline{t}, C) : C$ (C fresh variable)

$$\frac{\kappa_i \rightsquigarrow \gamma_i \ : \sigma_i}{(\kappa_1, \ldots, \kappa_n) \rightsquigarrow (\gamma_1, \ldots, \gamma_n) : (\sigma_1, \ldots, \sigma_n)} \quad \text{(conjunctions)}$$

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

The above relation can now be used to transform a clause of p into a clause for the efficient unfolder p_u:

$$p(\bar{t}) \leftarrow \quad \leadsto_u \quad p_u(\bar{t}, true) \leftarrow \qquad \text{(facts)}$$

$$\frac{\kappa \leadsto \gamma : \sigma}{p(\bar{t}) \leftarrow \kappa \quad \leadsto_u \quad p_u(\bar{t}, \sigma) \leftarrow \gamma} \qquad \text{(rules)}$$

Given an annotation $\mathcal{A}$ we define $P_u^{\mathcal{A}} = \{c' \mid c \in P \wedge c \leadsto_u c'\}$.

The transformation $\leadsto_u$ also generates calls to p_m predicates which we define later. These predicates take care of the global control and also return a filtered and renamed version of the call to be specialised as their last argument.

In the above definition inserting a literal of the form $p_u(\bar{t}, C)$ corresponds to further *unfolding* whereas inserting $p_m(\bar{t}, C)$ corresponds to stopping the unfolding process and leaving the atom for the global control (something which is also referred to as *memoisation*). In the case of the program P from Example 3 with $\mathcal{A}$ as depicted in Figure 2, we get the following program $P_u^{\mathcal{A}}$:

```
nont_u(X,T,R,(V1,V2)) :- t_u(a,T,V,V1),nont_m(X,V,R,V2).
nont_u(X,T,R,V1) :- t_u(X,T,R,V1).
t_u(X,[X|R],R,true).
```

Let us suppose for the moment that `nont_m` is defined by the following (i.e., it performs no global control nor does it filter and rename):

```
 nont_m(X,V,R,nont(X,V,R)).
```

Evaluating the above code for the call `nont_u(c,T,R,Leaves)` then yields two computed answers which correspond to the two branches in Figure 4:

```
> ?-nont_u(c,T,R,Leaves).
   T = [a|_A],
   Leaves = true,nont(c,_A,R) ? ;
   T = [c|R],
   Leaves = true ? ;
  no
```

3.3 The global control

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 leaf atoms of one branch of the SLDNF-tree, so we need to add some code that collects the information from all the branches. This can be done very efficiently using Prolog's `findall` predicate. It is here that we leave the purely declarative context, which we are entitled to do as our generating extensions do not have to be self-applied![11]

In essence, findall(V,Call,Res) finds all the answers θ_i of the call Call, applies θ_i to V and then instantiates Res to a list containing all the Vθ_i's. In particular, findall(B,nont_u(c,R,T,B),Bs) instantiates Bs to [[true,nont(c,_48,_49)], [true]]. This essentially corresponds to the leaves of the SLDNF-tree in Figure 4 (by flattening and removing the *true* atoms 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.

Now, once all the resultants have been generated, the body atoms have to be generalised (using gen_Δ) and then unfolded if they have not been encountered yet. This is achieved by re-defining the predicates p_m so that they perform the global control. 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 applying $filter_\Delta$; e.g., 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 already been encountered, and if not, unfolds the atom to produce the corresponding residual code.

We have the following definition of p_m (we denote the Prolog conditional by $If \rightarrow Then; Else$):

Definition 20. Let P be a program and p/n be a predicate defined in P. Also, let $\overline{v}$ a sequence of n distinct variables (one for each argument of p). We then define the clause $C_m^{p,\Delta}$ for p_m as follows:

```
pm(v̄,R) :-
   ( find_pattern(p(v̄),R) -> true
    ;  (generalise(p(v̄),p(ḡ)),
        insert_pattern(p(ḡ),Hd),
         findall(clause(Hd,Bdy),pu(ḡ,Bdy),Cs),
         pp(Cs),
         find_pattern(p(v̄),R) ) ).
```

Finally we define the Prolog program $P_m^\Delta = \{C_m^{p,\Delta} \mid p \in Pred(P)\}$.

In the above, the predicate find_pattern checks whether its first argument $p(\overline{v})$ is a call that has already been encountered and, if it is, its second argument will be instantiated to the properly renamed and filtered version $filter_\Delta(p(\overline{v}))$ of the call. This is achieved by keeping a list of the predicates that have been encountered before along with their renamed and filtered calls. Thus, if the call

[11] To stay declarative one would have to use something like the ground representation (see, e.g., [47, 41]), which would severely undermine our efficiency (and simplicity) concerns. This is why the *cogen* approach is more difficult to realise in a language like Gödel.

to find_pattern succeeds, then R has been instantiated to the residual call of $p(\overline{v})$, if not then the other branch of the conditional is executed.

The call generalise($p(\overline{v})$,$p(\overline{g})$) simply computes $p(\overline{g}) = gen_\Delta(p(\overline{v}))$. In some cases, this can be done beforehand (i.e., in the *cogen*) and does not have to feature in the generating extension. We will return to this issue below.

The predicate insert_pattern adds an new atom (its first argument $p(\overline{g})$) to the list of atoms already encountered and returns (in its second argument Hd) the renamed and filtered version $\mathit{filter}_\Delta(p(\overline{g}))$ of the atom. The atom Hd will then provide (maybe further instantiated) the head of the residual clauses. 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(clause(Hd,Bdy),p_u($\overline{g}$,Bdy),Cs) unfolds the generalised atom $p(\overline{g})$ and returns a list of residual clauses for $p(\overline{g})$ (in Cs). As we have seen in Section 3.2, the call to $p_u(\overline{g}, Bdy)$ inside this findall returns one leaf goal of the SLDNF-tree for $p(\overline{g})$ at a time and instantiates $p(\overline{g})$ (and thus also Hd) via the computed answer substitution of the respective branch. Observe that every atom in the leaf goal has already been renamed and filtered by a call to the corresponding m-predicate predicate.

The predicate pp pretty-prints the clauses of the residual program. The last call to find_pattern will instantiate the output argument R to the residual call $\mathit{filter}_\Delta(p(\overline{v}))$ of the atom $p(\overline{v})$.

We can now fully define what a generating extension is:

Definition 21. Let P be a program and $(U_\mathcal{A}, \Delta)$ a safe *BTC* for P, then the *generating extension* of P with respect to $(U_\mathcal{A}, \Delta)$ is the Prolog program $P_g = P_u^\mathcal{A} \cup P_m^\Delta$.

The generating extension is called as follows: if one wants to specialise an atom $p(\overline{v})$ one simply calls p_m($\overline{v}$,R). Observe generalisation and specialisation occur as soon as the we call p_m($\overline{v}$,R). Together with our particular construction of the unfolder predicates (Definition 19) this means that to ensure correctness of specialisation we need LD-safety instead of just safety (cf. Definition 10).

There are several improvements that one can add on top of this definition. The first improvement relates to the call generalise($p(\overline{v})$,$p(\overline{g})$) computing $p(\overline{g}) = gen_\Delta(p(\overline{v}))$. If the division for p in Δ is monovariant and simple (i.e., only contains static and dynamic) one can actually compute $p(\overline{g}) = gen_\Delta(p(\overline{v}))$ beforehand (i.e., in the *cogen* as opposed to in the generating extension), without having to know the actual values for the variables in $\overline{v}$! This will actually be used by our *cogen*, whenever possible, to further improve the efficiency of the generating extensions. For example, if we have $\Delta = \{p(\mathit{static}, \mathit{dynamic})\}$ and if $p(\overline{v})$ = p(X,Y) then we can simply put $p(\overline{g})$ = p(X,Z), where Z is a fresh variable. Thus, when executing the generating extension the static value in X will be kept and the dynamic value in Y will be abstracted.

Second, in practice it might be unnecessary to define p_m for every predicate p. Indeed, there might be predicates which are always reducible and are also never specialised by the user. Such predicates will never appear at the global level, and one can safely remove the corresponding definitions for p_m from Definition 21.

For instance, in Example 3 the predicate $t/3$ is always reducible and never specialised immediately by the user. Also, the division is monovariant and simple, and one can thus pre-compute `generalise`. The resulting, optimised generating extension is shown in Figure 7.

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

Figure 7: The generating extension for the parser

3.4 The *cogen*

The job of the *cogen* is now quite simple: given a program P and a safe *BTC* β for P, produce 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 A. 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.

3.5 An Example

We now show that our *cogen* approach is actually powerful enough to satisfactorily specialise the vanilla metainterpreter (a task which has attracted a lot of attention [14, 54, 71] and is far from trivial).

Example 4. The following is the well-known vanilla metainterpreter for the non-ground representation:

```
demo(true).
demo((P & Q)) :- demo(P),demo(Q).
demo(A) :- dclause(A,Body), demo(Body).

dclause(append([],L,L),true).
dclause(append([H|X],Y,[H|Z]),append(X,Y,Z) & true).
dclause(dapp(X,Y,Z,R), (append(X,Y,I) & (append(I,Z,R)&true))).
```

Note that in a setting with just static/dynamic one cannot specialise this program in an interesting way, because the argument to demo may (and usually will) contain variables. This is why neither [33] nor [57] were able to handle this example.

We, however, can produce the *BTC* $(\mathcal{A}, \Delta)$ with $\Delta = \{demo(nonvar), dclause(nonvar, dynamic)\}$ and where the annotation $\mathcal{A}$ is such that everything but the $demo(P)$ call in the second clause is marked as reducible. The demo_u unfolder predicate generated by the *cogen* for demo then looks like:

```
demo_u(true,true).
demo_u(B & C,(D,E)) :- demo_m(B,D), demo_u(C,E).
demo_u(F,(G,H)) :- dclause_u(F,I,G), demo_u(I,H).
```

The specialised code that is produced by the generating extension for the call $demo(dapp(X, Y, Z, R))$ is:

```
demo__0(B,C,D,E) :- demo__1(B,C,F), demo__1(F,D,E).
demo__1([],B,B).
demo__1([C|D],E,[C|F]) :-  demo__1(D,E,F).
```

Observe that specialisation has been successful: all the overhead has been compiled away and demo__1 even corresponds to the definition of append! Given the above *BTC*, our *cogen* can achieve a similar feat for *any* object program and query to be specialised! As we will see in Section 5 it can even do so at blinding speed.

4 Extending the *cogen*

In this section we will describe how to extend the *cogen* to handle elements that are necessary to handle logic programming languages with built-in's and non-declarative features. We will explain these extensions for Prolog, but many of the ideas should also carry over to other logic languages. (Proponents of Mercury and Gödel may safely skip all but Subsection 4.1.)

4.1 Declarative primitives

It is straightforward to extend the *cogen* to handle declarative primitives, i.e. built-ins such as `=/2`, `is/2` and `arg/3`,[12] or externally defined user predicates (as long as these are declarative). The code of these predicates is not available to the *cogen* and therefore no predicates to unfold them can be generated. The generating extension can therefore do one of two things:

1. either completely evaluate a call to such primitives (reducible case),
2. or simply produce a residual call (non-reducible case).

To achieve this, we simply extend the transformation of Definition 19 with the following two rules, where c is a call to a declarative primitive and reducible calls are underlined:

$$\underline{c} \rightsquigarrow c : true$$

$$c \rightsquigarrow true : c$$

Example 5. For instance, we have $\underline{\texttt{arg(1,X,A)}} \rightsquigarrow \texttt{arg(1,X,A)} : true$, meaning that the call will be executed in the generating extension and nothing has to be done in the specialised program. On the other hand, we have $\texttt{arg(N,X,A)} \rightsquigarrow true : \texttt{arg(N,X,A)}$, meaning that the call is only executed within the specialised program. Now take the clause:

$\texttt{p(X,N,A) :- }\underline{\texttt{arg(1,X,A)}}\texttt{,arg(N,X,A).}$

This clause is transformed (by $\rightsquigarrow_u$) into the following unfolding clause:

```
p_u(X,N,A,arg(N,X,A)):- arg(1,X,A).
```

For $\Delta = \{p(static, dynamic, dynamic\}$ and for `X = f(a,b)` the generating extension will produce the residual code:

```
p__0(N,a) :- arg(N,f(a,b),a).
```

while for $X = a$ the call `arg(1,a,A)` will fail and no code will be produced (i.e., failure has already been detected within the generating extension).

4.2 Problems with non-declarative primitives

The above two rules could also be used for non-declarative primitives. However, the code generated will in general be incorrect, for the following two reasons.

First, for some calls c to non-declarative primitives $c, fail$ is not equivalent to $fail$. For example, `print(a),fail` behaves differently from `fail`. Predicates for which the equivalence $p(\bar{t}), fail \equiv fail$ does not hold are termed as "side-effect" in [64]. For such predicates the independence on the selection rule does not hold. In the context of the Prolog left-to-right selection rule, this means that we have to ensure that failure to the right of such a call c does not prevent the

[12] E.g., `arg/3` can be viewed as being defined by a series facts: `arg(1,h(X),X).`, `arg(1,f(X,Y),X).`, `arg(2,f(X,Y),Y).`, ...

generation of the residual code for c nor its execution at runtime. For example, the clause

```
    t :- print(a), 2=3.
```

can be specialised to `t :- print(a),fail.` but not to `t :- fail,print(a).` (code has been generated for `print(a)` but we have changed the selection rule), neither to `t :- fail.` nor to the empty program. The scheme of Section 4.1 would produce the following unfolder predicate, which is incorrect as it produces the empty program:

```
    t_u(print(a)) :- 2=3.
```

The second problem are the so called "propagation sensitive" [64] built-in's. For calls c to such built-in's, even though $c, \mathit{fail} \equiv \mathit{fail}$ holds, the equivalence $(c,\ X = t) \equiv (X = t,\ c)$ does not. One such built-in is `var/1`: we have, e.g., `(var(X),X=a)` $\not\equiv$ `(X=a,var(X))`. Again, independence on the selection rule is violated (even though there are no side-effects), which again poses problems for specialisation. Take for example the following clause:

```
    t(X) :- var(X), X=a.
```

The scheme of Section 4.1 would produce the following unfolder predicate:

```
    t_u(X,var(X)) :- X=a.
```

Running this for `X` uninstantiated will produce the following residual code, which is incorrect as it always fails:

```
    t(a) :- var(a).
```

To solve this problem we will have to ensure that bindings generated by specialising calls to the right of propagation sensitive calls c do not backpropagate [64, 62] onto c. In the case above, we have to prevent the binding `X/a` to backpropagate onto the `var(X)` call.

In the remainder of this section we show how side-effect and propagation sensitive predicates can be dealt with in a rather elegant and still efficient manner in our *cogen* approach.

4.3 Hiding failure and sensitive bindings

To see how we can solve our problems, we examine a small example in more detail. Take the following program:

```
    p(X) :- print(X),var(X), q(X).
    q(a).
```

We have that $\underline{\texttt{q(X)}} \rightsquigarrow$ `q_u(X,C)` : `C`, and applying the scheme from Section 4.1 naively, we get:

```
   p_u(X,(print(X),var(X),C)) :- q_u(X,C).
   q_u(a,true).
```

For the same reasons as in the above examples this unfolder predicate is incorrect (e.g., for `X=b` the empty program is generated).

To solve the problem we have to avoid that the bindings generated by `q_u(X,C)` backpropagate onto `print(X),var(X)` and ensure that a failure of `q_u(X,C)` does not prevent code being generated for `print(X)`. One solution is to wrap q_u(X,C) into a call to findall. Such a call will not instantiate q_u(X,C) and if q_u(X,C) fails this will only lead to the third argument of findall being instantiated to an empty list. A first attempt might thus look like this:

```
  p_u(X,(print(X),var(X),Cs)) :-  findall(C,q_u(X,C),Cs).
  q_u(a,true).
```

If we now run p_u(X,Code) we get the residual code:

```
   p(X) :- print(X),var(X),true.
```

So, backpropagations have been prevented but unfortunately there is also now no link at all between the instantiations performed by q_u(X,C) and the rest of the clause.

To arrive at a full solution we have to re-create this link within the residual code (but *not* within the generating extension). If we knew that q_u(X,C) would always have exactly one solution (which it has not; it can fail) we could write the following code:

```
  p_u(X,(print(X),var(X),X=Xs,Cs)) :-
        findall((X,C),q_u(X,C),[(Xs,Cs)]).
  q_u(a,true).
```

Notice how we have added the variables of q(X) as an extra argument to the findall and how X=Xs re-creates the link between the variables in Cs and the rest of the clause. Running p_u(X,Code) produces the correct residual code:

```
   p(X) :- print(X),var(X),X=a,true.
```

The general solution, now, is to use an auxiliary predicate make_disjunction which transforms the result of the findall into the necessary residual code (e.g., it transforms [(a,true)] into X=a,true, [] into fail, or something like [(a,true),(Y,ground(Y))] into (X=a,true) ; (X=Y ; ground(Y))). All this leads to the following extra rule, to be added to Definition 19, and where calls whose bindings and whose failure should be hidden are wrapped into a hide_nf annotation:

$$\frac{\kappa \rightsquigarrow \gamma : \sigma}{\begin{array}{c} \mathit{hide\_nf}(\kappa) \rightsquigarrow \\ \texttt{varlist}(\kappa, V), \\ \texttt{findall}((\sigma, V), \gamma, R), \\ \texttt{make\_disjunction}(R, V, C) \\ : C \end{array}} \qquad R, V, C \text{ fresh variables}$$

The full code of make_disjunction is straightforward and can be found in Appendix A.

One might wonder why in the above solution one just keeps track of the variables in κ. The reason is that all the variables in γ or σ but not in κ are fresh, existential. Thus, these variables cannot occur in the remainder of the clause.

Note that annotating a call c using hide_nf also prevents right-propagation of bindings generated while specialising c. This is not a restriction, because instead of writing **hide_nf**(α),β we can always write **hide_nf**((α,β)) if one wants the instantiations of α to be propagated onto β. Furthermore, preventing right-propagations will turn out to be useful in the treatment of negations, conditionals, and disjunctions below.

Example 6. Let us trace the thus extended *cogen* on another example:

```
p(X) :- print(X), q(X).
q(a).
q(b).
```

If we mark `q(X)` as reducible and wrap it into a **hide_nf()** annotation, we get the following unfolding predicate for `p`:

```
p_u(X,(print(X),Disj)) :-
    varlist(q(X),Vars),
    findall((Code,Vars), q_u(X,Code), Cs),
    make_disjunction(Cs,Vars,Disj).
```

If we run the generating extension we get the residual program (calls to **true** have been removed by the *cogen*):

```
p__0(B) :- print(B), (B = a ; B = b).
```

4.4 Generating correct annotations

Having solved the problem of left-propagation of failure and bindings, we now just have to figure out when `hide_nf` are actually necessary. In order to achieve maximum specialisation and efficiency, one would want to use just the minimum number of such annotations which still ensures correctness. (To further improve specialisation and efficiency one could also introduce additional annotations such as `nf`(κ) if only non-failing has to be prevented and **hide**(κ) if only bindings have to be hidden. This is actually done within the implementation of the *cogen*, but, for clarity's sake, we don't elaborate on this here.)

The following modified rule for conjunctions (replacing the corresponding rule in Definition 19) ensures that no bindings are left-propagated or side-effects removed.

$$\frac{\kappa_i \leadsto \gamma_i : \sigma_i \;\wedge\; impure(\kappa_i) \Rightarrow \forall j > i : \models_{hide} \gamma_j}{(\kappa_1, \ldots, \kappa_n) \leadsto (\gamma_1, \ldots, \gamma_n) : (\sigma_1, \ldots, \sigma_n)}$$

Here $impure(\kappa_i)$ holds if κ_i contains a call to side-effect predicate (which has to be non-reducible) or to a non-reducible propagation sensitive call. Calls are classified as in [64] (e.g., the property of generating a side-effect propagates up the dependency graph).

We also define a new relation $\models_{hide} \gamma$ that holds if the code γ within the generating extension cannot fail and cannot instantiate variables in the remainder

$$\frac{}{\models_{hide} true}$$

$$\frac{\models_{hide} \gamma_i}{\models_{hide} \gamma_1, \ldots, \gamma_n}$$

$$\frac{\models_{hide} \gamma_i}{\models_{hide} (\gamma_1 \rightarrow \gamma_2; \gamma_3)}$$

$$\frac{\models_{hide} \gamma_i}{\models_{hide} (\gamma_1; \gamma_2)}$$

$$\frac{hide\_nf(\kappa) \leadsto \gamma : \sigma}{\models_{hide} \gamma}$$

Figure 8: The non-fail $\models_{nf}$ and hide relations $\models_{hide}$

of the generating extension. This relation is defined in Figure 8. This definition can actually be kept quite simple because it is intended to be applied to code in the generating extension which has a very special form.

Together with the modified rule for conjunctions, Figure 8 can be used to determine the required **hide_nf** annotations. For example, the first rule in Figure 8 actually implies that non-reducible calls never pose a problem and do not have to be wrapped into a **hide_nf** annotation.

4.5 Negation and Conditionals

Prolog's negation (not/1) is handled similar to a declarative primitive, except that for the residual case $not(\kappa)$ we will also specialise the code κ inside the negation and we have to make sure that this specialisation (performed by generating extension) cannot fail (otherwise the code generation would be incorrectly prevented) or propagate bindings.

$$\frac{\kappa \leadsto \gamma : true}{\underline{\texttt{not}}(\kappa) \leadsto \texttt{not}(\gamma) : true}$$

$$\frac{\kappa \leadsto \gamma : \sigma \quad \wedge \quad \models_{hide} \gamma}{\texttt{not}(\kappa) \leadsto \gamma : \texttt{not}(\sigma)}$$

The first rule is used when κ can be completely unfolded and it can be determined whether it fails or not: if γ succeeds then the generating extension will not generate code, and if γ fails the generating extension will succeed and produce the residual code `true` for the negation.

If the negation is non-reducible then we require that the subgoal in the generating extension does not fail (which can be ensured if necessary, by adding hide_nf annotations).

Example 7. Consider the following two annotated clauses.

$$\begin{array}{lll}\texttt{p(X)} & \texttt{:-} & \underline{\texttt{not}}\texttt{(X}\underline{\texttt{=}}\texttt{a).}\\ \texttt{q(Y)} & \texttt{:-} & \texttt{not(Y=a).}\end{array}$$

In the first clause X is assumed to be of binding-type static (or at least not nonvar) so the negation can be reduced.[13] In the second we assume that Y is dynamic. If we run the generating extension with goal p(a) we will get an empty program, which is correct. If we run the generating extension with goal q(Y) we will get the following (correct) residual clause:

```
  q__0(B) :- not(B=a).
```

Handling conditionals is also straightforward. If the test goal of a conditional is reducible then we can evaluate the conditional within the generating extension. If the test goal of the conditional is non-reducible then, similarly to the negation, we require that the three subgoals in the generating extension do not fail nor propagate bindings:

$$\frac{\kappa_i \leadsto \gamma_i : \sigma_i}{(\kappa_1\underline{\texttt{->}}\kappa_2\underline{\texttt{;}}\kappa_3) \leadsto (\gamma_1\texttt{->}(\gamma_2,\sigma_2\texttt{=}C)\texttt{;}(\gamma_3,\sigma_3\texttt{=}C)) : C} \quad \text{(C fresh variable)}$$

$$\frac{\kappa_i \leadsto \gamma_i : \sigma_i \quad \wedge \quad \models_{hide} \gamma_i}{(\kappa_1\texttt{->}\kappa_2\texttt{;}\kappa_3) \leadsto \gamma_1,\gamma_2,\gamma_3 : (\sigma_1\texttt{->}\sigma_2\texttt{;}\sigma_3)}$$

4.6 Disjunctions

The disjunction is not a non-declarative feature in itself, as it is just a shorthand notation for a set of ordinary clauses. So, one may at first think that the rule for disjunctions should be similar to the rule for conjunctions, but there is an important difference. Each goal κ_i in a disjunction $\kappa_1;..;\kappa_n$ gives rise to a goal γ_i in the generating extension that will compute the residual code of κ_i. The generating extension will then contain code $\gamma_1,..,\gamma_n$ that will compute the residual code for the components of the disjunction. If some γ_i fails for some i then this would cause the whole specialisation of the disjunction to fail, but this would in most cases be wrong, since other parts of the disjunction may succeed. This means that we have to ensure that the code in the generating extension that specialises each disjunct of a disjunction cannot fail, in much the same manner as we handled side-effect primitives.

[13] Note that it is up to the binding-type analysis to mark negations as reducible only if this is sound, e.g., when the arguments are ground.

Similarly, if one γ_i instantiates some variable X then this must not influence the value of X in some other γ_j (or even outside of the conjunction, as different bindings could be given to X in the different conjuncts, thus resulting in failure of the generating extension) where $j \neq i$. This is exactly the same problem we faced for propagation sensitive primitives.

Both these problems can thus be easily solved by using our `hide_nf` annotation (which will also prevent right-propagation of bindings, as required for the disjunction). The rule for disjunctions therefore has the form:

$$\frac{\kappa_i \leadsto \gamma_i : \sigma_i \qquad \models_{hide}\ \gamma_i}{(\kappa_1;\ldots;\kappa_n) \leadsto (\gamma_1,\ldots,\gamma_n) : (\sigma_1;\ldots;\sigma_n)}$$

The above rule will result in a disjunction being created in the residual code. We could say that the disjunctions are residualised. It is possible to treat disjunction in a different way in which they are reduced away, but at the price of some duplication of work and residual code. The rule for such reducible disjunctions is:

$$\frac{\kappa_i \leadsto \gamma_i : \sigma_i}{(\kappa_1\underline{;}\ldots\underline{;}\kappa_n) \leadsto (\gamma_1, \sigma_1\texttt{=C};\ldots;\gamma_n, \sigma_n\texttt{=C}) : \texttt{C}} \quad (\texttt{C}\text{ fresh variable})$$

The drawback of this rule is that that it may duplicate work and code. To see this consider a goal of the form: $Q_h, (Q_1;Q_2), Q_t$. If specialisation of $Q_1;Q_2$ does not give any instantiation of the variables that occur in Q_h and Q_t then these will be specialised twice and identical residual code will be generated each time.

4.7 More refined treatment of the `call` predicate

In this section we present one example of specialisation using the `call` predicate and show how its specialisation can be further improved. The `call` predicate can be considered to be declarative[14] and is important for implementing higher-order primitives in Prolog. Unfortunately, current implementations of `call` are not very efficient and it would therefore be ideal if the overhead could be removed by specialisation. This is exactly what we are going to do in this section.

In `call(C)` the value of `C` can either be a call to a primitive or (user-defined) predicate. Unless the predicate is externally defined the two cases require different treatment. Consider the following example, featuring the Prolog implementation of the higher-order map predicate:

```
map(P,[],[]).
map(P,[H|T],[PH|PT]) :- Call =.. [P,H,PH],
    call(Call), map(P,T,PT).
inc(X,Y) :- Y is X + 1.
```

[14] If delayed until its argument is `nonvar`, it can be viewed as being defined by a series facts: `call(p(X)) :- p(X).`, `call(q(X,Y)) :- q(X,Y).`, ...

Assume that we want to specialise the call `map(inc,I,O)`. We can produce the BTC $(\mathcal{A}, \Delta)$ with $\Delta = \{\texttt{map}(\mathit{static}, \mathit{dynamic}, \mathit{dynamic}), \texttt{inc}(\mathit{dynamic}, \mathit{dynamic})\}$ and where $\mathcal{A}$ marks everything, but the `map` call in clause 2, as reducible.

Since the value of `Call` is not known when we generate the unfolding predicate for `map` there is no way we can unfold the atom bound to `Call`. The unfolding predicate generated by the *cogen* thus looks like:

```
map_u(B,[],[],true).
map_u(C,[D|E],[F|G],(H,I)) :-
  H =.. [C,D,F], map_m(C,E,G,I).
```

The specialised code obtained for the call `map(inc,I,O)` is:

```
map__0([],[]).
map__0([B|C],[D|E]) :- inc(B,D), map__0(C,E).
```

All the overhead of `call` and `=..` has been specialised away, but one still needs the original program to evaluate `inc`. To overcome this limitation, one can devise a special treatment for calls to user-defined predicates which enables unfolding *within* a `call/1` primitive:

$\underline{\mathit{call}}(A) \leadsto \texttt{add\_extra\_argument}(A, "_u", C, G), \mathit{call}(G) : C$ (C fresh variable)
$\mathit{call}(A) \leadsto \texttt{add\_extra\_argument}(A, "_m", C, G), \mathit{call}(G) : C$ (C fresh variable)

In both cases the argument to `call` has to be a user-defined predicate which will be known by the generating extension but is not yet known at *cogen* time. If this is not the case one has to use the standard technique for built-ins and possibly keep the original program at hand.

The code for `add_extra_argument` can be found in Appendix A. It is used to construct calls to the unfolder and memoisation predicates. For example, calling `add_extra_argument("_u",p(a),Code,C)` gives `C = p_u(a,C)`.

Using this more refined treatment, the *cogen* will produce the following unfolder predicate:

```
map_u(B,[],[],true).
map_u(C,[D|E],[F|G],(H,I)) :-
  J =..[C,D,F],
  add_extra_argument("_u",J,H,K), call(K),
  map_m(C,E,G,I).
```

The specialised code obtained for the call $map(inc, I, O)$ is then:

```
map__0([],[]).
map__0([B|C],[D|E]) :-  D is B + 1, map__0(C,E).
```

All the overhead of map has been removed and we have even achieved unfolding of `inc`.

In the case we know the length of the list, we can even go further and remove the list processing overhead! In fact, we can now produce the BTC $(\mathcal{A}, \Delta')$ with $\Delta' = \{map(static, list(dynamic), dynamic),\ inc(dynamic, dynamic)\}$. If we then specialise $map(inc, [X, Y, Z], O)$ we obtain the following:

```
map__0(B,C,D,[E,F,G]) :- E is B + 1, F is C + 1, G is D + 1.
```

5 Benchmark Results

In this section we present a series of detailed experiments with our *cogen* system as well as with some other specialisation systems.

A first study of the speed of the *cogen* approach was performed in [33]. However, due to the limitations of the initial *cogen* only very few realistic benchmarks could be run. In particular, most of the benchmarks of the DPPD suite [40] could not be used because they require the treatment of partially instantiated data. The improved *cogen* of this paper can now deal with all the benchmarks in [40]. We thus ran our system on a selection of benchmarks from [40]. To test the ability to specialise non-declarative built-in's we also devised one new non-declarative benchmark: specialising the non-ground unification algorithm with occurs-check from [68]. More detailed descriptions about all the benchmarks can be found in Appendix D.

The implementation of the new *cogen* is actually called LOGEN, runs under Sicstus Prolog and is publicly available. We compare the results of LOGEN with the latest versions of MIXTUS [64] (version 0.3.6) and ECCE [49, 15]. (Comparisons of the initial *cogen* with other systems such as LOGIMIX, PADDY, and SP can be found in [33]).

All the benchmarks were run under `SICStus Prolog 3.7.1` on a Sun Ultra E450 server with 256Mb RAM operating under `SunOS 5.6`.

A summary of all the transformation times can be found in Table 1. The times for MIXTUS contains the time to write the specialised program to file (as we are not the implementors of MIXTUS we were unable to factor this part out), as does the column marked "with" for ECCE. The column marked "w/o" is the pure transformation time of ECCE without measuring the time needed for writing to file. The times for LOGEN exclude writing to file. For LOGEN, the column marked by *cogen* contains the runtimes of the *cogen* to produce the generating extension, whereas the column marked by *genex* contains the times needed by the generating extensions to produce the specialised programs. To be fair, it has to be emphasised that the binding-type analysis was carried out by hand. In a fully automatic system thus, the column with the *cogen* runtimes will have to be increased by the time needed for the binding-type analysis. However, the binding-type analysis and the *cogen* have to be run only *once* for every program and division. Thus, the generating extension produced for *regexp.r*1 was re-used without modification for *regexp.r*2 and *regexp.r*2 while

Program	MIXTUS	ECCE		LOGEN	
	with	with	w/o	cogen	genex
ex_depth	200 ms	230 ms	190 ms	1.5 ms	7.2 ms
grammar	220 ms	200 ms	140 ms	6.5 ms	1.1 ms
map.rev	70 ms	60 ms	30 ms	2.7 ms	1.0 ms
map.reduce	30 ms	60 ms	30 ms	”	1.3 ms
match.kmp	50 ms	90 ms	40 ms	1 ms	2.5 ms
model_elim	460 ms	240 ms	170 ms	3 ms	3.1 ms
regexp.r1	60 ms	110 ms	80 ms	1.3 ms	1.4 ms
regexp.r2	240 ms	120 ms	80 ms	”	2.5 ms
regexp.r3	370 ms	160 ms	120 ms	”	10.2 ms
transpose	290 ms	190 ms	150 ms	1.2 ms	1.9 ms
ng_unify	2510 ms	na	na	5.3 ms	3.5 ms

Table 1: Specialisation Times

Program	Original	MIXTUS	ECCE	LOGEN
ex_depth	1470 ms	680 ms	540 ms	530 ms
	1	2.16	2.72	2.77
grammar	2880 ms	200 ms	300 ms	190 ms
	1	14.40	9.60	15.16
map.rev	230 ms	100 ms	150 ms	120 ms
	1	2.30	1.53	1.92
map.reduce	540 ms	180 ms	150	170 ms
	1	3.00	3.60	3.18
match.kmp	3740 ms	2570 ms	1940 ms	3260 ms
	1	1.46	1.93	1.15
model_elim	1210 ms	340 ms	320 ms	450 ms
	1	3.56	3.78	2.69
regexp.r1	3240 ms	520 ms	760 ms	510 ms
	1	6.23	4.26	6.35
regexp.r2	900 ms	360 ms	350 ms	300 ms
	1	2.50	2.57	3.00
regexp.r3	1850 ms	550 ms	590 ms	1610 ms
	1	3.36	3.14	1.15
transpose	1590 ms	70 ms	70 ms	70 ms
	1	22.71	22.71	22.71
ng_unify	1600 ms	360 ms	na	430 ms
	1	4.44	-	3.72

Table 2: Runtimes and speedups of the specialised programs

the one produced for *map.rev* was re-used for *map.reduce*. Note that ECCE can only handle declarative programs, and could therefore not be applied on the *ng_unify* benchmark.

As can be seen in Table 1, LOGEN is by far the fastest specialisation system overall, running up to almost 3 orders of magnitude faster than the existing online systems. And, as can be seen in Table 2, the specialisation performed by the LOGEN system is not very far off the one obtained by MIXTUS and ECCE; sometimes LOGEN even surpasses both of them (for *ex_depth*, *grammar*, *regexp.r*1 and *regexp.r*2)! Being a pure offline system, LOGEN cannot pass the KMP-test, which can be seen in the timings for *match.kmp* in Table 2. (To be able to pass the KMP-test, more sophisticated local control would be required, see [56] and the discussion below.) To be fair, both ECCE and MIXTUS are fully automatic systems guaranteeing termination, while for LOGEN further work will be needed so that the binding-type classifications used in the above benchmarks can be derived automatically (while still ensuring termination). We return to this issue below. Nonetheless, the LOGEN system is surprisingly fast and produces surprisingly good specialised programs.

Finally, the figures of LOGEN 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 [26]: generating the compiler generator takes about 100 hours (including garbage collection), generating a generating extension took for the examples in [26] 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 generating extensions (including pre- and post-processing) still range from 113s to 447s.

6 Discussion and Future Work

6.1 Developing an automatic *BTA*

The most obvious way to improve the LOGEN system presented in this paper is to develop an automatic binding-type analysis. Currently, annotations and divisions have to be done by hand. This allows the experienced user to have a good control over the specialisation performed, but it is still an error-prone process and is definitely out of the reach of users not familiar with partial evaluation.

One might think that a binding-type analysis would just boil down to type and mode inference, e.g., as done by Mercury [65]. Unfortunately, Mercury's type and mode system does not cater for *dynamic* arguments whose mode and (possibly) also type is not known. Also, there is a tight interaction between the binding-type classification and the annotation, meaning that existing groundness, type or mode analysis cannot be applied directly. Fortunately, a solution has been developed in [10], based upon abstract interpretation and termination analysis.

[10] follows the following approach: Given a program P to be analysed it transforms it into a program P', which, when executed, behaves as an on-line specialiser. The on-line specialiser is different from usual ones in the sense that it — like off-line specialisers — uses the availability of arguments to decide on the unfolding of calls and that its unfolding decisions are derived by a *termination analysis*. Next, *abstract interpretation* is applied to gather information about the run-time behaviour of the on-line specialiser to settle at analysis time all the unfolding decisions. This is done in such a manner as to ensure termination of the local control. (To ensure global termination, one would have to integrate techniques such as [23] developed for functional programs.) Experiments also indicate that the quality of the analysis is very good [10]. Also, in [10] a term satisfies a binding-type iff it is *rigid wrt a given semi-linear norm*. It can thus handle binding-types such as `static` (i.e., terms rigid wrt the termsize norm) but also more sophisticated binding-types such as list skeletons (i.e., terms rigid wrt the list-length norm). Any such type can be handled by our improved LOGEN system and the technique of [10] seems to be the ideal ingredient to turn LOGEN into a fully automatic system. So, in the (near) future we plan to extend the LOGEN system such that it incorporates (an extension of) the binding-time analysis developed in [10].

6.2 Related Work

The first hand-written compiler generator based on partial evaluation principles was, in all probability, the system *RedCompile* for a dialect of Lisp [3]. Since then successful compiler generators have been written for many different languages and language paradigms [63, 29, 30, 5, 1, 24].

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 [58, 59].[15] However, it is not based on (off-line) partial deduction.

Also the construction of our program $P_u^{\mathcal{A}}$ (Definition 19) is related to the idea of *abstract compilation* [27]. In abstract compilation a program P is first transformed and abstracted. Running this transformed program then performs the actual abstract interpretation analysis of P.[16] In our case concrete execution of $P_u^{\mathcal{A}}$ performs (part of) the partial deduction process. Another similar idea has also been used in [69] to calculate abstract answers. Finally, [22] uses a transformation similar to ours to compute trace terms for the global control of logic and functional program specialisation (however, the specialisation technique itself is still basically online).

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

[16] In [12], a different kind of abstract compilation is presented, in which the transformed programs are analysed (but do not perform the analysis by themselves).

The local control component of our generating extensions is still rather limited: either a call is always reducible or never reducible. To remedy this problem, and to allow any kind of partially instantiated data, an extension of our cogen approach has been developed in [56]. This approach uses a sounding analysis (at specialisation time) to measure the minimum depth of partially instantiated terms. The result of this analysis is then used to control the unfolding and ensure termination. This approach allows more aggressive unfolding than the technique presented in this paper, passing the KMP-test and rivalling online systems in terms of flexibility. Due to the sounding analysis, however, it is not fully offline. In terms of speed of the specialisation process, it is hence slower than our fully offline cogen approach (but still much faster than online systems such as MIXTUS or ECCE). Also, [56] only addresses the local control component and it is still unclear how it can be extended for the global control (the prototype in [56] uses the online ECCE system for global control; to this end a trace terms were built up in the generating extension like in [22]).

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, for example, 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 [37] 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.

6.3 Semi-Online Treatment

Some built-in's can be treated in a more refined fashion than described in Section 4. For instance, for a call `var(X)` which is non-reducible we could still check whether the call fails or succeeds in the generating extension. If the call fails, we know that it will definitely fail at runtime as well. In that case we don't have to generate code and we thus achieve improved specialisation over a purely offline approach. If the call `var(X)` succeeds, however, we have gained nothing and still have to perform `var(X)` at runtime.

Similarly, for a call such as `ground(X)`, if it succeeds in the generating extension we can simply generate `true` in the specialised program. In that case we have again improved the efficiency of the specialised program. If, on the other hand, `ground(X)` fails in the generating extension it might still succeed at runtime: we have to generate the code `ground(X)` and have gained nothing.

The rules below cater for such a semi-online treatment of some built-in's.

$$\begin{array}{ll} \underline{c} \rightsquigarrow c : c & \text{if } c = \texttt{var}(t), \texttt{copy\_term}(s,t), s\texttt{\textbackslash==}t, \ldots \\ \underline{c} \rightsquigarrow (c \to C = true; C = c) : C & \text{if } c = \texttt{ground}(t), \texttt{nonvar}(t), \\ & \qquad \texttt{atom}(t), \texttt{integer}(t), s\texttt{==}t, \ldots \end{array}$$

The code of the *cogen* in Appendix A uses these optimisations if a `semicall` annotation is used (these annotations have not been used for the benchmarks in Section 5). It also contains a semi-online conditional, which reduces the conditional to the then branch (respectively else branch) if the test definitely succeeds (respectively definitely fails) in the generating extension.

Similarly, one can also produce a new binding-type, called `semi`, which lies in between `static` and `dynamic`. Basically, `semi` behaves like `static` for the generalisation gen_Δ (Definition 14) but like `dynamic` for filtering $filter_\Delta$ (Definition 16). The former means that an argument marked as `semi` will not be abstracted away by gen_Δ, while the latter allows such an argument to contain variables. Again, the code for these improvements can be found in Appendix A.

Another worthwhile improvement is to enable *semi-online* unfolding of predicates. In other words, instead of either always or never unfolding a predicate, one would like to either unfold the predicate or not based upon some (simple) criterion. This improvement is also very easy to achieve, and even requires no change to the *cogen* itself, just to the annotation process. Indeed, instead of marking a call $p(\bar{t})$ either as reducible or non-reducible we simply insert a static conditional into the annotated program: ($Test$ `->` $\underline{p}(\bar{t})$ `;` $p(\bar{t})$). Thus, if $Test$ succeeds the generating extension will unfold the call, otherwise it will be memoised.

We have actually used these improvements to produce a semi-online annotation of the *match.kmp* benchmark from Section 5. The results of this experiment (after some very simple post-processing) is as follows.

Program	cogen	genex	spec. runtime	speedup
match.kmp	1.2 ms	3.7 ms	2480 ms	1.51 ×

Note that LOGEN now outperforms MIXTUS, passes the KMP-test (actually, even without the post-processing; see [66]), and all that with incredible transformation speed.

6.4 More Future Work

In addition to extending the LOGEN to a fully automatic system, one might also think of extending its capabilities and domain of application.

First, one could try to extend the *cogen* approach so that it can achieve multi-level specialisation à la [24]. One could also try to use the *cogen* for run time code generation. A first version of the latter has in fact already been implemented; this actually does not require all that many modifications to our *cogen*. The former also seems to be reasonably straightforward to achieve.

One might also investigate whether the *cogen* approach can be ported to other logical programming languages. 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 not going to be straightforward to adapt the *cogen* approach for Gödel or Mercury [65] so that it still produces efficient generating extensions. Further work will be needed to establish this.

Finally, it also seems natural to investigate to what extent more powerful control techniques (such as characteristic trees [21, 49], trace terms [22] ore the local control of [56]) and specialisation techniques (like conjunctive partial deduction [46, 25, 15]) can be incorporated into the *cogen*, while keeping its advantages in terms of efficiency.

6.5 Conclusion

In the present paper we have formalised the concept of a *binding-type analysis*, allowing the treatment of *partially static* structures, in a (pure) logic programming setting and how to obtain a generic algorithm for offline partial deduction from such an analysis. We have then developed the *cogen* approach for offline specialisation, reaping the benefits of self-application without having to write a self-applicable specialiser. The resulting system, called LOGEN, is surprisingly compact and can handle partially static datastructures, declarative and non-declarative built-ins, disjunctions, conditionals, and the negation. We have shown that the resulting system achieves extremely fast specialisation, making it (to our knowledge) the fastest existing partial deduction system by far. We have also shown that the system can be applied on a wide range of natural benchmarks and that the resulting specialisation is also very good, sometimes even surpassing that of existing online systems.

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 providing valuable feedback and input on this paper.

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] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[3] 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.

[4] 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.

[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, Proceedings of PLILP'91*, LNCS 844, pages 198–214, Madrid, Spain, 1994. Springer-Verlag.

[6] R. Bol. Loop checking in partial deduction. *The 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] M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Proceedings of the European Symposium on Programming (ESOP'98)*, LNCS 1381, pages 27–41. Springer-Verlag, April 1998.

[11] W. Chen, M. Kifer, and D. S. Warren. A first-order semantics of higher-order logic programming constructs. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, pages 1090–1114. MIT Press, 1989.

[12] M. Codish and B. Demoen. Analyzing logic programs using "prop"-ositional logic programs and a magic wand. *The Journal of Logic Programming*, 25(3):249–274, December 1995.

[13] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'93)*, Charleston, South Carolina, January 1993. ACM Press.

[14] Y. Cosmadopoulos, M. Sergot, and R. W. Southwick. Data-driven transformation of meta-interpreters: A sketch. In H. Boley and M. M. Richter, editors, *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, volume 567 of *LNAI*, pages 301–308, Kaiserslautern, FRG, July 1991. Springer Verlag.

[15] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999. To appear.

[16] D. A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.

[17] 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.

[18] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[19] 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.

[20] 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.

[21] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[22] J. Gallagher and L. Lafave. Regular approximations of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 115–136, Schloß Dagstuhl, 1996. Springer-Verlag.

[23] A. J. Glenstrup and N. D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, LNCS. Springer-Verlag, June 25–28 1996.

[24] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. Swierstra and M. Hermenegildo, editors, *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, LNCS 982, pages 259–278, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[25] R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag.

[26] 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.

[27] 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.

[28] P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.

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

[30] C. K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. Working paper, 1992.

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

[32] 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.

[33] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag. Extended version as Technical Report CW 221, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~dtai`.

[34] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag.

[35] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.

[36] J. Lam and A. Kusalik. A comparative analysis of partial deductors for pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada, May 1990. Revised April 1991.

[37] J. Launchbury. *Projection Factorisations in Partial Evaluation.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.

[38] 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.

[39] 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, The Netherlands, September 1995. Springer-Verlag.

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

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

[42] 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.

[43] M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag.

[44] M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing*, 16:283–342, 1998.

[45] M. Leuschel and D. De Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *The Journal of Logic Programming*, 36(2):149–193, August 1998.

[46] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.

[47] M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 495–509, Portland, USA, December 1995. MIT Press.

[48] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Springer-Verlag.

[49] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[50] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 83–103, Stockholm, Sweden, August 1996. Springer-Verlag.

[51] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[52] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

[53] B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995.

[54] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.

[55] 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.

[56] J. Martin and M. Leuschel. Sonic partial deduction. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS, Novosibirsk, Russia, 1999. Springer-Verlag. To appear.

[57] 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.

[58] 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.

[59] 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.

[60] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.

[61] D. Poole and R. Goebel. Gracefully adding negation and disjunction to Prolog. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 635–641. Springer-Verlag, 1986.

[62] S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[63] 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.

[64] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

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

[66] M. H. Sørensen. *Introduction to Supercompilation*. LNCS. Springer-Verlag, 1998.

[67] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, December 1995. MIT Press.

[68] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[69] P. Tarau and K. De Bosschere. Memoing techniques for logic programs. In Y. Deville, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'93*, Workshops in Computing, pages 196–209, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.

[70] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[71] W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, pages 322–342, Leuven, Belgium, July 1997.

[72] E. Yardeni, T. Früwirth, and E. Shapiro. Polymorphically typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 63–90. MIT Press, 1992.

A A Prolog cogen

This appendix contains the listing of the cogen.

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

:- ensure_consulted('pp').

/* the file .ann contains:
   ann_clause(Head,Body),
   filter(Call,ListofBindingTypes),
   residual(P) */

reset_cogen :- reset_pp.

cogen :-
  findall(C,memo_clause(C),Clauses1),
  findall(C,unfold_clause(C),Clauses2),
  pp(Clauses1),
  pp(Clauses2).

flush_cogen :-
  print_header,
  flush_pp.

memo_clause(clause(Head,(find_pattern(Call,V) ->
                          true ;
                          (insert_pattern(GCall,H),
                           findall(NClause,
                                   (RCall, NClause = clause(H,Body)),
                                   NClauses),
                           pp(NClauses),
                           find_pattern(Call,V))
                        ) )) :-
  residual(Call),
  cogen_can_generalise(Call),
  generalise(Call,GCall),
  add_extra_argument("_u",GCall,Body,RCall),
  add_extra_argument("_m",Call,V,Head).

memo_clause(clause(Head,(find_pattern(Call,V) ->
                          true ;
                          (generalise(Call,GCall),
                           add_extra_argument("_u",GCall,Body,RCall),
                           insert_pattern(GCall,H),
                           findall(NClause,
                                   (RCall, NClause = clause(H,Body)),
                                   NClauses),
                           pp(NClauses),
                           find_pattern(Call,V))
                        ) )) :-
  residual(Call),
  not(cogen_can_generalise(Call)),
  add_extra_argument("_m",Call,V,Head).


unfold_clause(clause(ResCall,FlatResBody)) :-
  ann_clause(Call,Body),
  add_extra_argument("_u",Call,FlatVars,ResCall),
  body(Body,ResBody,Vars),
  flatten(ResBody,FlatResBody),
  flatten(Vars,FlatVars).

body((G,GS),GRes,VRes) :-
  body(G,G1,V),
  filter_cons(G1,GS1,GRes,true),
  filter_cons(V,VS,VRes,true),
  body(GS,GS1,VS).

body(unfold(Call),ResCall,V) :-
        add_extra_argument("_u",Call,V,ResCall).
body(memo(Call),AVCall,VFilteredCall) :-
        add_extra_argument("_m",Call,VFilteredCall,AVCall).

body(true,true,true).
body(call(Call),Call,true).
body(rescall(Call),true,Call).
body(semicall(Call),GenexCall,ResCall) :-
        specialise_imperative(Call,GenexCall,ResCall).

body(if(G1,G2,G3),    /* Static if: */
     ((RG1) -> (RG2,(V=VS2)) ; (RG3,(V=VS3))),
     V) :-
        body(G1,RG1,_VS1),
        body(G2,RG2,VS2),
        body(G3,RG3,VS3).
body(resif(G1,G2,G3), /* Dynamic if: */
     (RG1,RG2,RG3), /* RG1,RG2,RG3 shouldn't fail and be determinate */
     ((VS1) -> (VS2) ; (VS3))) :-
        body(G1,RG1,VS1),
        body(G2,RG2,VS2),
        body(G3,RG3,VS3).
body(semif(G1,G2,G3), /* Semi-online if: */
     (RG1,flatten(VS1,FlatVS1),
      ((FlatVS1 == true)
        -> (RG2,SpecCode = VS2)
        ;  ((FlatVS1 == fail)
            -> (RG3,SpecCode = VS3)
            ;  (RG2,RG3, (SpecCode = ((FlatVS1) -> (VS2) ; (VS3))))
           )
     )),
 /* RG1,RG2,RG3 shouldn't fail and be determinate */
      SpecCode) :-
        body(G1,RG1,VS1),
        body(G2,RG2,VS2),
        body(G3,RG3,VS3).

body(resdisj(G1,G2),(RG1,RG2),(VS1 ; VS2)) :- /* residual disjunction */
        body(G1,RG1,VS1),
        body(G2,RG2,VS2).
body( (G1;G2), ((RG1,V=VS1) ; (RG2,V=VS2)), V) :- /* static disjunction */
        body(G1,RG1,VS1),
        body(G2,RG2,VS2).

body(not(G1),    /* Static declarative not: */
     \+(RG1),true) :-
        body(G1,RG1,_VS1).
body(resnot(G1), /* Dynamic declarative not: */
     RG1,\+(VS1)) :-
        body(G1,RG1,VS1).

 body(hide_nf(G1),GXCode,ResCode) :-
       (body(G1,RG1,VS1)->
         (flatten(RG1,FlatRG1),
          flatten(VS1,FlatVS1),
            GXCode = (varlist(G1,VarsG1),
                      findall((FlatVS1,VarsG1),FlatRG1,ForAll1),
                      make_disjunction(ForAll1,VarsG1,ResCode)));
           (GXCode = true,
            ResCode=fail)).
 body(hide(G1),GXCode,ResCode) :-
       (body(G1,RG1,VS1)->
         (flatten(RG1,FlatRG1),
          flatten(VS1,FlatVS1),
            GXCode = (varlist(G1,VarsG1),
                      findall((FlatVS1,VarsG1),FlatRG1,ForAll1),
                      ForAll1 = [_|_], /* detect failure */
                      make_disjunction(ForAll1,VarsG1,ResCode)));
           (GXCode = true,
            ResCode=fail)).


/* some special annotations: */
body(ucall(Call),
        (add_extra_argument("_u",Call,V,ResCall),
         call(ResCall)),
        V).
body(mcall(Call),
        (add_extra_argument("_m",Call,V,ResCall),
         call(ResCall)),
        V).
body(det(Call),
        (copy_term(Call,CCall),
         ((CCall)->(C=(Call=CCall));(C=fail))),C).

make_disj([],fail).
make_disj([H],H) :- !.
make_disj([H|T],(H ; DT)) :-
        make_disj(T,DT).


make_disjunction([],_,fail).
make_disjunction([(H,CRG)],RG,FlatCode) :-
        !,simplify_equality(RG,CRG,EqCode),
        flatten((EqCode,H),FlatCode).
make_disjunction([(H,CRG)|T],RG,(FlatCode ; DisT)) :-
        simplify_equality(RG,CRG,EqCode),
        make_disjunction(T,RG,DisT),
        flatten((EqCode,H),FlatCode).

specialise_imperative(Call,Call,Call)
        :- varlike_imperative(Call),!.
specialise_imperative(Call,
        (Call -> (Code=true) ; (Code=Call)), Code) :-
         groundlike_imperative(Call),!.
specialise_imperative(X,true,X).

varlike_imperative(var(_X)).
varlike_imperative(copy_term(_X,_Y)).
varlike_imperative((_X\==_Y)).
groundlike_imperative(ground(_X)).
groundlike_imperative(nonvar(_X)).
groundlike_imperative(_X==_Y).
groundlike_imperative(atom(_X)).
groundlike_imperative(integer(_X)).


generalise(Call,GCall) :-
    ((filter(Call,ArgTypes),
      Call =.. [F|FArgs],
      l_generalise(ArgTypes,FArgs,GArgs))
    ->
       (GCall =..[F|GArgs])
    ;
```

```
     (print('*** WARNING: unable to generalise: '), print(Call),nl,
      GCall = Call)
    ).

cogen_can_generalise(Call) :-
    filter(Call,ArgTypes),
    static_types(ArgTypes). /* check whether we can filter at cogen time */

/* types which allow generalisation/filtering at cogen time */
static_types([]).
static_types([static|T]) :- static_types(T).
static_types([dynamic|T]) :- static_types(T).

generalise(static,Argument,Argument).
generalise(dynamic,_Argument,_FreshVariable).
generalise(free,_Argument,_FreshVariable).
generalise(nonvar,Argument,GenArgument) :-
    nonvar(Argument),
    Argument =.. [F|FArgs],
    make_fresh_variables(FArgs,GArgs),
    GenArgument =..[F|GArgs].
generalise((Type1 ; _Type2),Argument,GenArgument) :-
    generalise(Type1,Argument,GenArgument).
generalise((_Type1 ; Type2),Argument,GenArgument) :-
    generalise(Type2,Argument,GenArgument).
generalise(type(F),Argument,GenArgument) :-
    typedef(F,TypeExpr),
    generalise(TypeExpr,Argument,GenArgument).
generalise(struct(F,TArgs),Argument,GenArgument) :-
    nonvar(Argument),
    Argument =.. [F|FArgs],
    l_generalise(TArgs,FArgs,GArgs),
    GenArgument =..[F|GArgs].
generalise(semi,Argument,Argument). /* treat as static for generalisation */

l_generalise([],[],[]).
l_generalise([Type1|TT],[A1|AT],[G1|GT]) :-
    generalise(Type1,A1,G1),
    l_generalise(TT,AT,GT).

make_fresh_variables([],[]).
make_fresh_variables([_|T],[_|FT]) :-
    make_fresh_variables(T,FT).

typedef(list(T),(struct([],[]) ; struct('.',[T,type(list(T))]))).
typedef(model_elim_literal,(struct(pos,[nonvar]) ; struct(neg,[nonvar]))).


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).
```

```
filter_cons(H,T,HT,FVal) :-
        ((nonvar(H),H = FVal) -> (HT = T) ; (HT = (H,T))).

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

B The Parser Example

The annotated program looks like:

```
  /* file: parser.ann */
static_consult([]).

residual(nont(_,_,_)).
filter(nont(X,T,R),[static,dynamic,dynamic]).

ann_clause(nont(X,T,R), (unfold(t(a,T,V)),memo(nont(X,V,R)))).
ann_clause(nont(X,T,R), (unfold(t(X,T,R)))).

ann_clause(t(X,[X|Es],Es),true).
```

This supplies cogen with all the necessary information about the parser program, this is, the code of the program (with annotations) and the result of the binding-time analysis. The predicate `filter` defines the division for the program and the predicate `residual` represents the set $\mathcal{L}$ in the following way. If `residual`(A) succeeds for a call A then the predicate symbol p of A is in $Pred(P)\backslash\mathcal{L}$ and p is therefore one of the predicates for which a m-predicate is going to be generated. The annotations `unfold` and `memo` is used by cogen to determine whether or notq to unfold a call.

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

```
/* file: parser.gx */
/*  --------------------  */
/*  GENERATING EXTENSION  */
/*  --------------------  */
:- logen_reconsult('memo').
:- logen_reconsult('pp').

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),
      I = (clause(H,J))),K),
    pp(K),
    find_pattern(nont(B,C,D),E)
  )).
nont_u(B,C,D,',' (E,F)) :-
  t_u(a,C,G,E),
  nont_m(B,G,D,F).
nont_u(H,I,J,K) :-
  t_u(H,I,J,K).
t_u(L,[L|M],M,true).
```

Running the generating extension for

```
  nont(c,T,R)
```

yields the following residual program:

```
  nont__0([a|B],C) :-
    nont__0(B,C).
  nont__0([c|D],D).
```

Some other examples which can be handled by simple divisions (i.e., using just the binding-types **static** and **dynamic**), such as an interpreter for the ground representation (where the overhead is compiled away) and a "special" regular expression parser from [57] (where we obtain deterministic automaton after specialisation) can be found in [33].

C The Transpose Example

A possible annotated program of the transpose benchmark program for matrix transposition looks like:

```
  static_consult([]).

  residual(transpose(A,B)).
  filter(transpose(A,B),[type(list(type(list(dynamic)))),dynamic]).
  ann_clause(transpose(A,[]),unfold(nullrows(A))).
  ann_clause(transpose(A,[B|C]),
                (unfold(makerow(A,B,D)),unfold(transpose(D,C)))).

  filter(makerow(A,B,C),[type(list(type(list(dynamic)))),dynamic,dynamic]).
  ann_clause(makerow([],[],[]),true).
  ann_clause(makerow([[A|B]|C],[A|D],[B|E]),unfold(makerow(C,D,E))).

  filter(nullrows(A),[type(list(type(list(dynamic))))]).
  ann_clause(nullrows([]),true).
  ann_clause(nullrows([[]|A]),unfold(nullrows(A))).
```

In the above we stipulate that the first argument to transpose will be of type *list*(*list*(*dynamic*)), i.e., a list skeleton whose elements are in turn list skeletons (in other words we have a matrix skeleton, without the actual matrix elements). The generating extension produced by *cogen* then looks like this:

```
/* file: bench/transpose.gx */
/*  --------------------  */
/*  GENERATING EXTENSION  */
/*  --------------------  */
:- logen_reconsult('memo').
:- logen_reconsult('pp').

transpose_m(B,C,D) :-
  ((
    find_pattern(transpose(B,C),D)
   ) -> (
    true
   ) ; (
    generalise(transpose(B,C),E),
    add_extra_argument([95,117],E,F,G),
    insert_pattern(E,H),
    findall(I, (
      G,
      I = (clause(H,F))),J),
    pp(J),
    find_pattern(transpose(B,C),D)
  )).
transpose_u(B,[],C) :-
  nullrows_u(B,C).
transpose_u(D,[E|F],','(G,H)) :-
  makerow_u(D,E,I,G),
  transpose_u(I,F,H).
makerow_u([],[],[],true).
makerow_u([[J|K]|L],[J|M],[K|N],O) :-
  makerow_u(L,M,N,O).
nullrows_u([],true).
nullrows_u([[]|P],Q) :-
  nullrows_u(P,Q).
```

Running the generating extension in the LOGEN system leads to the following (and full unfolding has been achieved):

```
 =>transpose([[a,b],[c,d]],R).

filtered atom: transpose__0(a,b,c,d,_9441)

writing the specialised program to: bench/transpose.pe.transpose__

transpose__0(B,C,D,E,[[B,D],[C,E]]).
```

For the particular DPPD benchmark query used in Section 5 we actually had to use a sligthly more refined division:

```
filter(transpose(A,B),
 [(struct('[]',[]) ;
   struct('.',[type(list(dynamic)),type(list(dynamic))])),dynamic]).

filter(makerow(A,B,C),[type(list(type(list(dynamic)))),dynamic,dynamic]).
```

The above corresponds to giving the first argument of `transpose` the following binding-type (i.e., a list skeleton where only the first argument itself is also a list skeleton):

```
:- type arg1 --> [] ; [list(dynamic) | list(dynamic)].
```

D Benchmark Programs

The benchmark programs were carefully selected and/or designed in such a way that they cover a wide range of different application areas, including: pattern matching, databases, expert systems, meta-interpreters (non-ground vanilla, mixed, ground), and more involved particular ones: a model-elimination theorem prover, the missionaries-cannibals problem, a meta-interpreter for a simple imperative language. The benchmarks marked with a star (*) can be fully unfolded. Full descriptions of all but the unify benchmark can be found in [40].

Benchmark	Description
ex_depth	A variation of the *depth* Lam & Kusalik benchmark [36] with a more sophisticated object program.
grammar.lam	A Lam & Kusalik benchmark [36].
map.reduce	Specialising the higher-order map/3 (using call and =..) for the higher-order reduce/4 in turn applied to add/3.
map.rev	Specialising the higher-order map for the reverse program.
match.kmp	Try to obtain a KMP matcher. A benchmark based on the "match" Lam & Kusalik benchmark [36] but with improved run-time queries.
model_elim.app	Specialise the Poole-Goebel [61] model elimination prover (also used by De Waal-Gallagher [16]) for the append program.
regexp.r1	A naive regular expression matcher. Regular expression: (a+b)*aab.
regexp.r2	Same program as regexp.r1 for ((a+b)(c+d)(e+f)(g+h))*.
regexp.r3	Same program as regexp.r1 for ((a+b)(a+b)(a+b)(a+b)(a+b)(a+b))*.
transpose.lam*	A Lam & Kusalik benchmark [36].
unify	A non-ground unification algorithm with the occurs check, taken from page 152 of [68]. This program contains non-declarative built-in's such as `var/1`. The task is to specialise for `unify(f(g(a),a,g(a)),S)`.

Table 3: Description of the benchmark programs