Offline Specialisation in Prolog Using a Hand-Written Compiler Generator

MICHAEL LEUSCHEL

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

JESPER JØRGENSEN

Dept. of Mathematics and Physics, Royal Veterinary and Agricultural University,
Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark

WIM VANHOOF and MAURICE BRUYNOOGHE

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

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 the *cogen* approach is also applicable to the specialisation of logic programs (called partial deduction) and leads to effective specialisers. Moreover, using good binding-time annotations, the speed-ups of the specialised programs are comparable to the speed-ups obtained with online specialisers.

The paper first develops a generic approach to offline partial deduction and then a specific offline partial deduction method, leading to the offline system LIX for pure logic programs. While this is a usable specialiser by itself, it is used to develop the *cogen* system LOGEN. Given a program, a specification of what inputs will be static, and an annotation specifying which calls should be unfolded, LOGEN generates a specialised specialiser for the program at hand. Running this specialiser with particular values for the static inputs results in the specialised program. While this requires two steps instead of one, the efficiency of the specialisation process is improved in situations where the same program is specialised multiple times.

The paper also presents and evaluates an automatic binding-time analysis that is able to derive the annotations. While the derived annotations are still suboptimal compared to hand-crafted ones, they enable non-expert users to use the LOGEN system in a fully automated way.

Finally, LOGEN is extended so as to directly support a large part of Prolog's declarative and non-declarative features and so as to be able to perform so called mixline specialisations.

Keywords Partial evaluation, partial deduction, program specialisation, compiler generation, abstract interpretation

1 Introduction

Partial evaluation has over the past decade received considerable attention both

in functional (e.g. (Jones, Gomard and Sestoft 1993)), imperative (e.g. (Andersen 1994)) and logic programming (e.g. (Gallagher 1993, Komorowski 1992, Pettorossi and Proietti 1994)). Partial evaluators are also sometimes called *mix*, as they usually perform a mixture of evaluation and code generation steps. In the context of pure logic programs, partial evaluation is sometimes referred to as *partial deduction*, the term partial evaluation being reserved for the treatment of impure logic programs.

Guided by the *Futamura projections* (Futamura 1971) 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). In essence, given a particular program P, a *cogen* generates a *specialised specialiser* for P. If P is an interpreter a *cogen* thus generates a compiler.

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 (Sahlin 1993), or PADDY (Prestwich 1992)) is effectively self-applicable. On the other hand a partial deducer which specialises only purely declarative logic programs (like SAGE (Gurr 1994) or the system in (Bondorf, Frauendorf and Richter 1990)) 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 for logic programs have been obtained by (Fujita and Furukawa 1988) and (Mogensen and Bondorf 1992). However, the specialisation in (Fujita and Furukawa 1988) is incorrect with respect to some extra-logical built-ins, leading to incorrect results when attempting self-application (Bondorf et al. 1990). The partial evaluator LOGIMIX (Mogensen and Bondorf 1992) does not share this problem, but gives only modest speedups when self-applied (compared to results for functional programming languages; see (Mogensen and Bondorf 1992)) and cannot handle partially static data.

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 specialised specialisers. This is the background behind the approach to program specialisation called the *cogen approach* (as opposed to the more traditional *mix* approach): instead of trying to write a partial evaluation system *mix* 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

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

out to be just a simple extension of a "binding-time analysis" for logic programs (something first discovered for functional languages in (Holst 1989) and then exploited in, e.g., (Holst and Launchbury 1991, Birkedal and Welinder 1994, Andersen 1994, Glück and Jørgensen 1995, Thiemann 1996)).

In this paper we will describe the first *cogen* written in this way for a logic programming language. We start out with a *cogen* for a small subset of Prolog and progressively improve it to handle a large part of Prolog and to extend its capabilities.

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 procedure for *offline partial deduction* from such an analysis.
2. Based upon point 1, the first description of an efficient, handwritten compiler generator (*cogen*) for a logic programming language, which has — exactly as for other handwritten cogens for other programming paradigms — a quite elegant and natural structure.
3. A way to handle both *extra-logical* features (such as `var/1`) and *side-effects* (such as `print/1`) within the *cogen*. A refined treatment of the `call/1` predicate is also presented.
4. How to handle negation, disjunction and the if-then-else conditional in the *cogen*.
5. Experimental results showing the efficiency of the *cogen*, the generating extensions, and also of the specialised programs.
6. A method to obtain a binding-type analysis through the exploitation of existing termination analysers.

This paper is a much extended and revised version of (Jørgensen and Leuschel 1996): points 3, 4, 5, 6 and the partially static structures of point 1 are new, leading to a more powerful and practically useful *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. In Section 4 we discuss the treatment of declarative and non-declarative built-ins as well as constructs such as negations, conditionals, and disjunctions. In Section 5 we present experimental results underlining the efficiency of the *cogen* and of the generating extensions it produces. We also compare the results against a traditional offline specialiser. In Section 6 we

present a method for doing an automatic binding-type analysis. We evaluate the efficiency and quality of this approach using some experiments. We conclude with some discussions of related and future work in Section 7.

2 Off-line Partial Deduction

Throughout this paper, we suppose familiarity with basic notions in logic programming. We follow the notational conventions of (Lloyd 1987). In particular, in programs, we denote variables through strings starting with 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

We start off by presenting a general procedure for performing partial deduction. More details on partial deduction and how to control it can be found, e.g., in (Leuschel and Bruynooghe 2002).

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, non-trivial but possibly incomplete SLDNF-trees. (A *trivial* SLDNF-tree is one in which no literal in the root has been selected for resolution, while 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.) 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 non-failing branch. These SLDNF-trees and resultants are obtained as follows.

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

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 (Lloyd and Shepherdson

1991), 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 most practical approaches independence is ensured by using a *renaming* transformation which maps dependent atoms to new predicate symbols. Adapted correctness results can be found in (Benkerimi and Hill 1993, Leuschel, Martens and De Schreye 1998) and (Leuschel, De Schreye and de Waal 1996). Renaming is often combined with argument *filtering* to improve the efficiency of the specialised program; see e.g. (Gallagher and Bruynooghe 1990, Benkerimi and Hill 1993, Leuschel and Sørensen 1996).

Closedness can be ensured by using the following outline of a partial deduction procedure, similar to the ones used in e.g. (Gallagher 1991, Gallagher 1993, Leuschel and De Schreye 1998).

Procedure 1 (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: $\mathcal{S}_{new} := \mathit{generalise}(\mathcal{S}_0)$
repeat
  $\mathcal{S}_{old} := \mathcal{S}_{new}$
  $S_{new} := \{s_n \mid s_n \in \mathit{leaves}(\mathit{unfold}(P, s_o)) \wedge\ s_o \in \mathcal{S}_{old}\}$
  $S_{new} := \mathit{generalise}(\mathcal{S}_{old} \cup S_{new})$
until $\mathcal{S}_{old} = \mathcal{S}_{new}$ (modulo variable renaming)
output $\mathcal{S} := \mathcal{S}_{new}$

The above procedure is parametrised by an unfolding rule *unfold* and an generalisation operation *generalise*. The latter can be used to ensure termination and can be formally defined as follows.

Definition 3
An *generalisation operation* is a function *generalise* from sets of atoms to sets of atoms such that, for any finite set of atoms S, $\mathit{generalise}(S)$ is a finite set of atoms S' using the same predicates as those in S, and every atom in S is an instance of an atom in S'.

If Procedure 1 terminates then the closedness condition is satisfied. Finally, note that, two sets of atoms S_1 and S_2 are said to be identical *modulo variable renaming* if for every $s_1 \in S_1$ there exists $s_2 \in S_2$ such that s_1 and s_2 are variants, and vice versa.

2.2 Off-Line Partial Deduction and Binding-Types

In Procedure 1 one can distinguish between two different levels of control. The unfolding rule U controls the construction of the incomplete SLDNF-trees. This is called the *local control* (Gallagher 1993, Martens and Gallagher 1995). The generalisation 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 a description 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 (Consel and Danvy 1993, Jones et al. 1993) has mainly stressed off-line approaches, while supercompilation of functional (Turchin 1986, Sørensen and Glück 1995) and partial deduction of logic programs (Gallagher and Bruynooghe 1991, Sahlin 1993, Bol 1993, Bruynooghe, De Schreye and Martens 1992, Martens and De Schreye 1996, Martens and Gallagher 1995, Leuschel et al. 1998, De Schreye, Glück, Jørgensen, Leuschel, Martens and Sørensen 1999) have mainly concentrated on on-line control.

An initial motivation for using the off-line approach was to achieve effective self-application (Jones, Sestoft and Søndergaard 1989.). But the off-line approach is in general also much more efficient since many decisions concerning control are made *before* and not during specialisation. This is especially true in a setting where the same program is re-specialised several times. (Note, however, that the global control is usually not done in a fully offline fashion: almost all offline partial evaluators maintain during specialisation a list of calls that have been previously specialised or are pending (Jones et al. 1993).)

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*. 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 (Mogensen and Bondorf 1992) 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

This approach is successful for functional programs, but often proves to be too weak for logic programs: in logic programming partially instantiated data structures 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 (Hill and Gallagher 1998, Martens and De Schreye 1995) or most of the benchmarks from the DPPD library (Leuschel 1996-2000).

The basic idea to improve upon the above shortcoming, is to describe parts of arguments which will actually be known at specialisation time by a special form of

types.[2] Below, we will develop the first such description, of what we call *binding-types*, in logic programming.

Binding-Types

In logic programming, a type can be defined as a set of terms closed under substitution (Apt and Marchiori 1994). We will stick to this view and adapt the definitions and concepts of (Yardeni, Frühwirth and Shapiro 1992) (which mainly follow the Hilog notation (Chen, Kifer and Warren 1989)).

As is common in polymorphically typed languages (e.g. (Somogyi et al. 1996)), types are 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 presuppose the existence of three 0-ary type constructors: `static`, `dynamic`, and `nonvar`. These constructors will be given a pre-defined meaning below. Also, a type which contains no variables is called *ground*.

Definition 4
A *type definition* for a type constructor c of arity n is of the form

$$c(V_1, \ldots, V_n) \longrightarrow f_1(T_1^1, \ldots, T_1^{n_1}) \ ; \ \ldots \ ; \ f_k(T_k^1, \ldots, T_k^{n_k})$$

with $k \geq 1, n, n_1, \ldots, n_k \geq 0$ and where $f_1, \ldots, f_k$ are distinct function symbols, $V_1, \ldots, V_n$ are distinct type variables, and T_i^j are types which only contain type variables in $\{V_1, \ldots, V_n\}$.
A *type system* Γ is a 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 type system Γ_1, defining a type constructor for parametric lists, can be defined as follows: $\Gamma_1 = \{list(T) \longrightarrow nil \ ; \ cons(T, list(T))\}$. Using the ASCII notations of Mercury (Somogyi et al. 1996) and using Prolog's list notation, the type system Γ_1 would be written down as follows:

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

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.

We now define type judgements relating terms to types in the underlying type system Γ.

Definition 5

[2] This is somewhat related to the way instantiations are defined in the Mercury language (Somogyi, Henderson and Conway 1996). But there are major differences, which we discuss later.

Type judgements have the form $t : \tau$, where t is a term and τ is a type, and are inductively defined as follows.

- $t : dynamic$ holds for any term t
- $t : static$ holds for any ground term t
- $t : nonvar$ holds for any non-variable term t
- $f(t_1, \ldots, t_n) : c(\tau'_1, \ldots, \tau'_k)$ holds if there exists a ground instance of the type definition $Def_\Gamma(c)$ in the underlying type system Γ which has the form $c(\tau'_1, \ldots, \tau'_k) \longrightarrow \ldots f(\tau_1, \ldots, \tau_n) \ldots$ and where $t_i : \tau_i$ for $1 \leq i \leq n$.

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

Note that our definitions guarantee that types are downwards-closed in the sense that for all terms t and types τ we have $t : \tau \Rightarrow t\theta : \tau$.

Here are a few examples, using the type system Γ_1 above. First, we have $s(0) : static$, $s(0) : nonvar$, and $s(0) : dynamic$. Also, $s(X) : nonvar$, $s(X) : dynamic$ but not $s(X) : static$. For variables we have $X : dynamic$, but neither $X : static$ nor $X : nonvar$. A few examples with lists (using Prolog's list notation) are as follows: $[\,] : list(static)$, $s(0) : static$ hence $[s(0)] : list(static)$, $X : dynamic$ and $Y : dynamic$ hence $[X, Y] : list(dynamic)$. Finally, we have, for example, that $list(dynamic)$ is more general than $list(static)$.

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 (Jørgensen and Leuschel 1996)). For that we first define the concept of a division which assigns types to arguments of predicates.

Definition 6
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 $Pred(P)$, with at most one division for any predicate. 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 Δ is called *more general* than another division Δ' iff $\forall\, p(\tau'_1, \ldots, \tau'_n) \in \Delta'$ 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.

The fact that divisions only use ground types means that we do not cater for polymorhpic types, although we can still use parametric types. This simplifies the remainder of the presentation (mainly Definition 14) but can probably be lifted. As can be seen from the above definition, we restrict ourselves to monovariant divisions in this paper. As discussed in (Jones et al. 1993), a way to handle polyvariant divisions by a monovariant approach is to "invent sufficiently many versions of each predicate."

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 single

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 $\mathcal{U}$. 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 7
A *binding-type analysis* (*BTA*) yields, given a program P and an arbitrary 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 $\mathcal{S}_0$ of Procedure 1) can take. The rôle of Δ is to give information about the atoms and their binding types that can occur at the global level (i.e., the ones in $\mathcal{S}_{new}$ and $\mathcal{S}_{old}$ of Procedure 1). 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 $\mathcal{S}_{new}$ of Procedure 1 (given the restrictions imposed by the annotation of the *BTA*) corresponds to the patterns described by Δ.[3]

We first define a safety notion for atoms and goals.

Definition 8
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 $\exists p(\tau_1, \ldots, \tau_n) \in \Delta$ such 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 $\leftarrow p(a, a), p(b, c)$ are safe wrt $\Delta = \{p(static, dynamic)\}$ while $p(X, a)$ is not.

Definition 9
Let $\beta = (\mathcal{U}, \Delta)$ be a *BTC* for a program P. Then β is a *globally safe* BTC *for* P iff for every goal G which is safe wrt Δ, $\mathcal{U}(P, G)$ is an SLDNF-tree τ for $P \cup \{G\}$ whose leaf goals are safe wrt Δ. A *BTA* is *globally safe* if for any program P it produces a globally safe *BTC* for P.

Sometimes — in order to simplify both the partial deducer and the *BTA* — one might want to generalise atoms and then lift them to the global level (i.e., $\mathcal{S}_{new}$ in Procedure 1) *before* the full SLDNF-tree τ has been built, namely at the point where a left-to-right selection rule would have selected the atom. This is the motivation behind the following notion of a *strongly* globally safe *BTC*.

[3] Our safety condition differs somewhat from the classical *uniform congruence* requirement (Launchbury 1991, Jones et al. 1993). We discuss this difference in Section 7.1.

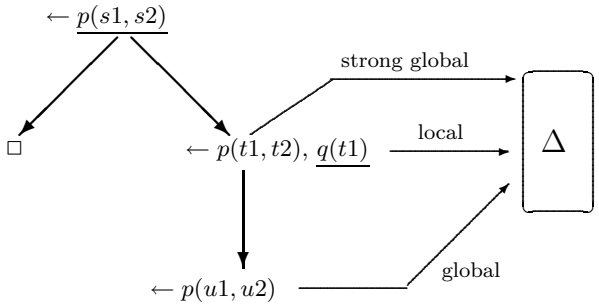

Fig. 1. Different types of safety for a sample, incomplete SLD-tree

Definition 10
Let $\beta = (\mathcal{U}, \Delta)$ be a *BTC* for a program P. Then β is a *strongly globally safe BTC* for P iff it is globally safe for P and for every goal G which is safe wrt Δ, $\mathcal{U}(P, G)$ is an SLDNF-tree such that the literals to the left of selected literals are also safe wrt Δ.

Notice, that in the above definitions of safety no requirement is made about the actual atoms selected by $\mathcal{U}$. Indeed, contrary to functional or imperative programming languages, definite logic programs can handle uninstantiated variables and a positive atom can always be selected. Nonetheless, if we have negative literals or Prolog built-ins, 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 calls to built-ins 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, $\mathcal{U}(P, G)$ is an SLDNF-tree for $P \cup \{G\}$ where all selected literals are safe wrt Δ.

The difference between local and global safety is illustrated in Figure 1. Note that it might make sense to use different divisions for local and global safety. This can be easily allowed, but we will not do so in the presentation of this article.

Let us now return to the global control. Definition 9 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 9 to ensure safety at all stages of Procedure 1, the particular generalisation 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 generalisation operation *generalise* is *safe wrt a division* Δ iff for every finite set of atoms S which is safe wrt Δ, $generalise(S)$ is also safe wrt Δ .

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

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 globally and locally safe *BTC* for P. E.g., the goal $\leftarrow app([a, b], Y, Z)$ is safe wrt Δ and $\mathcal{U}$ 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 (and thus static), leads only to goals whose first arguments are ground (static).

2.3 LIX*, 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).

We first define a particular class of simple-minded but effective unfolding rules.

Definition 13
An *annotation* $\mathcal{A}$ for a program P marks every literal in the body of each clause of P as either *reducible* or *non-reducible*. A program P together with an annotation $\mathcal{A}$ for P is called an *annotated program*, and is denoted by $P_{\mathcal{A}}$.
Given an annotation $\mathcal{A}$ for P, $U_{\mathcal{A}}$ denotes the unfolding rule which given a goal G computes $U_{\mathcal{A}}(P, G)$ by unfolding the leftmost atom in G and then continously unfolds leftmost reducible atoms until an SLD-tree is obtained with only non-reducible atoms in the leaves.

Syntactically we represent an annotation for P by underlining the predicate symbol of reducible literals.[4]

[4] 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

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

Let $\Delta = \{p(static), q(static, dynamic)\}$. Then $\beta = (U_{\mathcal{A}}, \Delta)$ is a globally 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, hence 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 *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 *BTAs* will thus return results of the form $\beta = (U_{\mathcal{A}}, \Delta)$.

Given we have a BTC for a program P, in order to arrive at a concrete instance of Procedure 1 we now only need a (safe) generalisation operation, which we define in the following.

Definition 14
We first define a family of mappings gen_τ from terms to terms, parameterised by types, inductiely as follows:

- $gen_{static}(t) = t$, 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
- $gen_{c(\tau'_1,\ldots,\tau'_k)}(f(t_1,\ldots,t_n)) = f(gen_{\tau_1}(t_1),\ldots,gen_{\tau_n}(t_n))$, if there exists a ground instance in $Def_\Gamma(c)$ of the form $c(\tau'_1,\ldots,\tau'_k) \longrightarrow \ldots; f(\tau_1,\ldots,\tau_n); \ldots$.

Let Δ be a division for some program P. We then define the partial mapping gen_Δ from atoms to atoms by:

- $gen_\Delta(p(t_1,\ldots,t_n)) = p(gen_{\tau_1}(t_1),\ldots,gen_{\tau_n}(t_n))$ if $\exists\ p(\tau_1,\ldots,\tau_n) \in \Delta$ such that $p(t_1,\ldots,t_n) : p(\tau_1,\ldots,\tau_n)$.

We also define the generalisation operation $generalise_\Delta$ as follows: For a set S of atoms which is safe w.r.t. Δ, $generalise_\Delta(S)$ is a minimal subset S_1 of $S_2 = \{gen_\Delta(s) \mid s \in S\}$ such that for every element s of S_2 there exists a variant of s in S_1.

For example, if $\Delta = \{p(static, dynamic),\ q(dynamic, static, nonvar)\}$ we have $gen_\Delta(p(a,b)) = p(a,X)$ and $gen_\Delta(q(a,b,f(c))) = q(Y,b,f(Z))$. We also have that $generalise_\Delta(\{p(a,b),\ q(a,b,f(c))\}) = \{p(a,X), q(Y,b,f(Z))\}$.
For $\Delta' = \{r(list(dynamic))\}$ (where $list(dynamic)$ 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 because it is not safe w.rt. Δ.

As can be seen gen_Δ is in general not total, but is total for atoms safe wrt Δ. Hence, in the context of a globally safe *BTA*, gen_Δ and $generalise_\Delta$ will always be defined.

Proposition 1
For every division Δ, $generalise_\Delta$ is safe wrt Δ.

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

Definition 15
Let $\|.\|$ be a fixed mapping from atoms to natural numbers such that $\|A\| = \|B\|$ iff A and B are variants. We then define $filter_\Delta$ as follows: $filter_\Delta(A) = p_{\|gen_\Delta(A)\|}(V_1\theta, \ldots, V_k\theta)$, where $A = gen_\Delta(A)\theta$, p is the predicate symbol of A, and $V_1, \ldots, V_k$ are the variables appearing in $gen_\Delta(A)$.

The purpose of the mapping $\|.\|$ is to assign to every specialised atom (i.e., atoms of the form $gen_\Delta(A)$) a unique identifier and predicate name, thus ensuring the independence condition (Lloyd and Shepherdson 1991). The $filter_\Delta$ operation will properly rename instances of these atoms and also filter out static parts, thus improving the efficiency of the residual code (Gallagher and Bruynooghe 1990, Benkerimi and Hill 1993). 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 that $filter_\Delta(p(a, b)) = p_1(b)$ as well as $filter_\Delta(q(a, b, f(c))) = q_2(a, c)$.

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

Procedure 2 (off-line partial deduction)

1. Perform a globally safe *BTA* (possibly by hand) returning results of the form $(U_\mathcal{A}, \Delta)$.
2. Perform Procedure 1 with $U_\mathcal{A}$ as unfolding rule and $generalise_\Delta$ as generalisation operation. The initial set of atoms $\mathcal{S}_0$ should only contain atoms which are safe wrt Δ.
3. Construct the specialised program P' using $filter_\Delta$ and the output $\mathcal{S}$ of Procedure 1 as follows: $P' = \{filter_\Delta(A)\theta \leftarrow filter_\Delta(B_1), \ldots, filter_\Delta(B_n) \mid A\theta \leftarrow B_1, \ldots, B_n \in resultants(U_\mathcal{A}(P, A)) \wedge A \in \mathcal{S}\}$.

Proposition 2
Let $(U_\mathcal{A}, \Delta)$ be a globally 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 Procedure 2 are safe wrt Δ.

Notably, if Procedure 2 terminates then the final set $\mathcal{S}$ will be safe wrt Δ. However, none of our notions of safety actually ensure (local or global) termination of Procedure 2. Termination is thus another issue (orthogonal to safety) which a *BTA* has to worry about. Basically, the annotation $\mathcal{A}$ has to be such that for all atoms A which are safe wrt Δ, $U_\mathcal{A}$ returns a finite SLDNF-tree τ for $P \cup \{\leftarrow A\}$. Furthermore, Δ has to be such that $generalise_\Delta$ ensures that only finitely many atoms can appear at the global level. We will return to this issue in Section 6.

We now illustrate Procedure 2 on a relatively simple example.

Example 3
We use a small generic parser for a set of languages which are defined by grammars of the form $N ::= aN|X$ (where a is a terminal symbol and X is a placeholder for a terminal symbol). The example is adapted from (Komorowski 1992) and the (annotated) parser P is depicted in Figure 2. The first argument to *nont* is the value for X while the other two arguments represent the string to be parsed as a difference list.

1. Given the initial division $\Delta_0 = \{nont(static, dynamic, dynamic)\}$, a BTA might return $\beta = (U_{\mathcal{A}}, \Delta)$ with $\Delta = \{nont(static, dynamic, dynamic), t(static, dynamic, dynamic)\}$ and where $\mathcal{A}$ is represented in Figure 2. It can be seen that β is a globally and locally safe BTC for P.
2. Let us now perform the proper partial deduction for $\mathcal{S}_0 = \{nont(c, T, R)\}$. Note that the atom $nont(c, T, R)$ is safe wrt Δ_0 (and hence also wrt Δ). Unfolding the atom in $\mathcal{S}_0$ yields the SLD-tree in Fig. 3. We see that the only atomin the leaves is $\{nont(c, V, R)\}$ and we obtain $\mathcal{S}_{old} = \mathcal{S}_{new}$ (modulo variable renaming).
3. The specialised program before and after filtering is depicted in Figure 4. Note that, if one wishes to call the filtered version in exactly the same way as the unfiltered one has to add the clause $nont(c, T, R) \leftarrow nont_1(T, R)$.

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

Fig. 2. A very simple parser

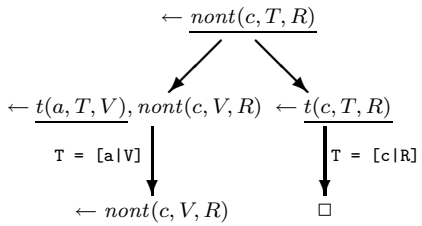

Fig. 3. Unfolding the parser of Figure 2

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

Fig. 4. Unfiltered and filtered specialisation of Figure 2

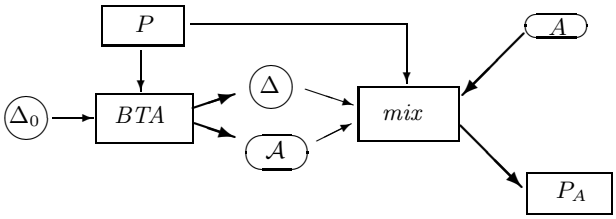

Fig. 5. Overview of the *mix* approach

The LIX *system*

Based upon Procedure 2 we have implemented a concrete offline partial deduction system called LIX using the traditional *mix* approach (Jones et al. 1993) depicted in Figure 5. We will examine the power of this system in more detail in Section 5. As we will see, provided that a good *BTA* is used, the quality of the specialised code provided by LIX can be surprisingly good. As is to be expected, due to its offline nature, LIX itself is very fast. In the next section, we show how the specialisation speed can be further improved by using the *cogen* approach.

Now, a crucial aspect for the performance of LIX is of course the quality of the *BTC*. Also, the runtime of an automatic *BTA* can usually not be neglected, and it could be considerably higher than that of LIX. However, in cases where the same code is specialised over and over again, the cost of the *BTA* is much less significant, as it only has to be run once. We will return to these issues in Sections 5 and 6.

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 the 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 receives as its only input an atom A which is safe wrt Δ, which it then specialises (using parts 2 and 3 of Procedure 2 with $\mathcal{S}_0 = \{A\}$), thereby producing a specialised program P_A. In the particular context of Example 3 a generating extension is a program that, when given the safe atom $nont(c, T, R)$, produces the residual program shown in Figure 4.

In this section, we develop the *compiler generator* LOGEN; it is a program that given a program P and a globally 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 6 (the κ, γ, and σ subscripts will be explained in the next section), and also shows the differences with the more traditional *mix* approach presented in Figure 5. As can be seen, P, Δ, and $\mathcal{A}$ have been compiled into the generating extension $genex^{P}_{\mathcal{A},\Delta}$ (contributing to

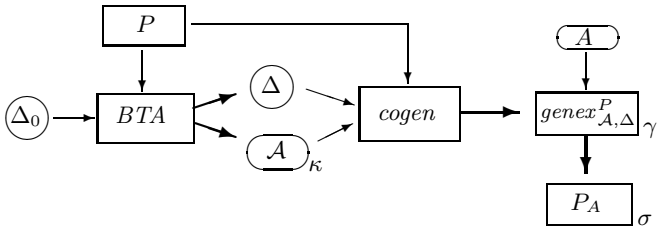

Fig. 6. Overview of the *cogen* approach

its efficiency and also making it standalone). A generating extension is thus not a generic partial evaluator, but a highly specialised one: it can specialise the program P only for calls A which are safe wrt Δ and it can only follow the annotation $\mathcal{A}$.

To explain and formalise the *cogen* approach, we will first examine the rôle and structure of generating extensions $genex^P_{\mathcal{A},\Delta}$. Once this is clear we will consider how the *cogen* can generate them.

3.2 The local control

The crucial idea for simplicity (and efficiency) of the generating extensions is to produce a specific "unfolding" predicate p_u for each predicate p/n. Also, for every predicate which is susceptible to appear at the global level, we will produce a specific "memoisation" predicate p_m.

Let us first consider the local control aspect. This predicate p_u 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 rule $U_{\mathcal{A}}$ to $p(t_1, ..., t_n)$, whereby it will return in B the atoms in the leaf of the branch and also instantiate $t_1, ..., t_n$ via the composition of *mgu*s of the branch (see Figure 7). 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 can be collected in the last argument B.

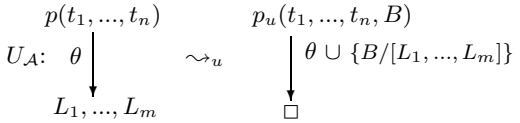

Fig. 7. Going from p to p_u

All this can be obtained by transforming every clause for p/n into a clause for

$p_u/(n+1)$ in the following manner. To simplify the presentation, we from now on use the notation $p(\overline{t})$ to represent an atom of the form $p(t_1, \ldots, t_n)$ and also $p(\overline{s}, t)$ to represent an atom of the form $p(s_1, \ldots, s_n, t)$.

We first define the ternary relation $\kappa \rightsquigarrow \gamma : \sigma$. Intuitively (see Figure 6), $\kappa \rightsquigarrow \gamma : \sigma$ denotes that the *cogen* will produce from the annotated literal or conjunction κ in the original program P the calls γ in the generating extension $genex^P_{\mathcal{A},\Delta}$. In turn, the computed answers of γ will instantiate σ to the bodies of the residual clauses that are part of the specialised program P_S. 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.

Definition 16
The ternary relation $\kappa \rightsquigarrow \gamma : \sigma$, with κ denoting annotated conjunctions, γ denoting conjunctions and σ denoting terms, is defined by the following three rules. Remember that an underlined literal is selected for unfolding.

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

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

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

The above relation can now be used to define the relation $\rightsquigarrow_u$ which transforms a clause of p into a clause for the efficient *unfolder* p_u.

$$p(\overline{t}) \leftarrow \quad \rightsquigarrow_u \quad p_u(\overline{t}, \mathit{true}) \leftarrow \qquad \text{(facts)}$$

$$\frac{\kappa \rightsquigarrow \gamma : \sigma}{p(\overline{t}) \leftarrow \kappa \quad \rightsquigarrow_u \quad p_u(\overline{t}, \sigma) \leftarrow \gamma} \qquad \text{(rules)}$$

Given an annotation $\mathcal{A}$ and a program P we define $P^{\mathcal{A}}_u = \{c' \mid c \in P \wedge c \rightsquigarrow_u c'\}$.

Note that the transformation $\rightsquigarrow_u$, by means of the $\rightsquigarrow$ transformation, 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 $p_u(\overline{t}, C)$ corresponds to further *unfolding* whereas inserting $p_m(\overline{t}, C)$ corresponds to stopping local unfolding 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 $P^{\mathcal{A}}_u$, where (V_1, V_2) and V_1 represent σ of Definition 16:

```
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).
```

Suppose for the moment the simplest definition possible for `nont_m` (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 3 and allow us to reconstruct the unfiltered specialisation 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

As mentioned, the above code for $P_u^{\mathcal{A}}$ is still incomplete, and we have to extend it to perform the global control 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 easily using Prolog's `findall` predicate.[5]

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 renamings of all the `V`θ_i's. In particular, `findall(B,nont_u(c,T,R,B),Bs)` instantiates `Bs` to `[[true,nont(c,_48,_49)], [true]]`. This essentially corresponds to the leaves of the SLDNF-tree in Figure 3 (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 get in `Cs` a representation of the two resultants of Figures 3 and 4 (without filtering).

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 $p(\overline{t})$ in the original program, if one calls $p_m(\overline{t}, R)$ then R will be instantiated to the residual call of $p(\overline{t})$ (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 17
Let P be a program and p/n be a predicate defined in P. Also, let $\overline{v}$ be 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:

[5] Note that, because our generating extensions do not have to be self-applied, we do not necessarily have to specialise the `findall` predicate itself.

```
p_m(v̄,R) :- ( find_pattern(p(v̄),R) -> true
            ;  (generalise(p(v̄),p(ḡ)),
                insert_pattern(p(ḡ),Hd),
                findall(clause(Hd,Bdy),p_u(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 an instance of a call that has already been specialised (or is in the process of being specialised) 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 the classical "seen before" check of partial evaluation (Jones et al. 1993) and 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 to `find_pattern` succeeds, then `R` has been instantiated to the residual call of $p(\overline{v})$, if the call was not seen before then the other branch of the conditional is executed.

The call `generalise(`$p(\overline{v})$`,`$p(\overline{g})$`)` simply computes $p(\overline{g}) = gen_\Delta(p(\overline{v}))$.

The predicate `insert_pattern` adds a 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 $filter_\Delta(p(\overline{g}))$ of the generalised atom. The atom `Hd` will 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 $filter_\Delta(p(\overline{g}))$ (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 $q(\overline{v})$ in the leaf goal has already been renamed and filtered by a call to the corresponding predicate $q_m(\overline{v})$.

Finally, the predicate `pp` pretty-prints the clauses of the residual program and the last call `find_pattern` will instantiate the output argument `R` to the residual call $filter_\Delta(p(\overline{v}))$ of the atom $p(\overline{v})$ (which is different from `Hd` which is $filter_\Delta(p(\overline{g}))$).

We can now fully define what a generating extension is:

Definition 18

Let P be a program and $(U_\mathcal{A}, \Delta)$ a strongly globally 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 that generalisation and specialisation occur *as soon as* we call $p_m(\overline{v}$,`R`$)$, and not after the whole incomplete SLDNF-tree has

been built.[6] Together with our particular construction of the unfolder predicates (Definition 16) this means that to ensure correctness of specialisation we need to have strong global safety instead of just global safety (cf. Definitions 9 and 10).

There are several ways to improve the definition of a generating extension (Definition 18). The first improvement relates to the call `generalise(`$p(\overline{v})$`,`$p(\overline{g})$`)` which computes $p(\overline{g}) = gen_\Delta(p(\overline{v}))$. If the division for p in Δ is 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 $\Delta = \{p(static, dynamic)\}$ and $p(\overline{v})$ = `p(X,Y)`, then the *cogen* does not have to generate a call to `generalise/2`; it can simply use `p(X,Z)` for $p(\overline{g})$, where Z is a fresh variable, within the code for `p_m(X,Y,R)`. The generating extension will thus correctly keep the static values in `X` and abstract the dynamic values in `Y`.

Second, in practice it might be unnecessary to define p_m for every predicate p. Indeed, there might be predicates which are never memoised. Such predicates will never appear at the global level, and one can safely remove the corresponding definitions for p_m from Definition 18.

For instance, in Example 3 the predicate $t/3$ is always reducible and never specialised immediately by the user. Also, the division is simple, and one can thus pre-compute `generalise`. The resulting, optimised generating extension is shown in Figure 8.

```
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).
```

Fig. 8. The generating extension for the parser

3.4 The *cogen* LOGEN

The job of the *cogen* is now quite simple: given a program P and a strongly globally 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*, called LOGEN,

[6] It is, however, not very difficult to change the *cogen* so that it calls $p_m(\overline{v}$,R) only after the whole incomplete SLDNF-tree has been built.

is shown in Appendix A. The predicate `memo_clause` generates the definition of the global control m-predicates for each non-reducible predicate of the program whereas the predicates `unfold_clause` 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 `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 LOGEN is actually powerful enough to satisfactorily specialise the vanilla metainterpreter (a task which has attracted a lot of attention (Cosmadopoulos, Sergot and Southwick 1991, Martens and De Schreye 1996, Vanhoof and Martens 1997) and is far from trivial).

Example 4
The following is the well-known vanilla metainterpreter for the non-ground representation, along with an encoding of the "double append" program:

```
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 the static/dynamic binding types one cannot specialise this program in an interesting way, because the argument to `demo` may (and usually will) contain variables. This is why neither (Jørgensen and Leuschel 1996) nor (Mogensen and Bondorf 1992) 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 every literal but the $demo(P)$ call in the second clause is marked as reducible (see underlining above).

Observe that, to make the BTA simpler, we encode conjunctions in a list-like fashion within the second argument of *dclause* as follows: a conjunction $A_1 \wedge \ldots \wedge A_n$ will be represented as $A_1 \& (\ldots (A_n \& true))$. This enables us to separate the conjunction skeleton from the individual literals, and allows us to produce an annotation which will result in removing all the parsing overhead related to the conjunction skeleton but will not unfold potentially recursive literals within the conjunctions.

The importance of the *nonvar* annotation is its influence on the generalisation operation. Indeed, we have $gen_{\Delta}(demo(append(X, [a], Z))) = demo(append(X, Y, Z))$ whereas for $\Delta' = \{demo(dynamic),\ dclause(dynamic,\ dynamic)\}$ the generalisation operation throws away too much information: $gen_{\Delta'}(demo(append(X, [a], Z)))$ $= demo(C)$, resulting in very little specialisation.

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 (after flattening) 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, LOGEN can achieve a similar feat for *any* object program and query to be specialised. As we will see in Section 5 it can do so efficiently.

Finally, note that the inefficiency of traversing the first argument to *dapp* twice has not been removed. For this, conjunctive partial deduction is needed (De Schreye et al. 1999).

4 Extending LOGEN

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

4.1 Declarative primitives

It is straightforward to extend LOGEN to handle declarative primitives, i.e. built-ins such as `=/2`, `is/2` and `arg/3`,[7] or externally defined user predicates (i.e., predicates defined in another file or module,[8] 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 16 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

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

[8] Of course, doing a modular binding-time analysis is more difficult than doing an ordinary one, but it is possible (Vanhoof 2000) and this is not really our concern here.

For instance, we have $\underline{\texttt{arg(1,X,A)}} \leadsto \texttt{arg(1,X,A)} : \mathit{true}$, 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 $\texttt{arg(N,X,A)} \leadsto \mathit{true} : \texttt{arg(N,X,A)}$, meaning that the call is only executed within the specialised program. Now take the clause:

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

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

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

For $\Delta = \{p(\mathit{static}, \mathit{dynamic}, \mathit{dynamic}\}$ 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).

Observe that, while `arg(1,a,A)` fails in SICStus Prolog, it actually raises an error in ISO Prolog. So, in the latter case we actually have to generate a residual clause of the form `p__0(N,A) :- raise_exception(...).`

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 p for which the conjunctions $p(\overline{t}), \mathit{fail}$ and fail are not equivalent are termed as "side-effect" in (Sahlin 1993). For such predicates the independence on the computation rule does not hold. In the context of the Prolog left-to-right computation rule, this means that we have to ensure that failure to the right of such a call c does not prevent the 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).` and 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" (Sahlin 1993) built-ins. For calls c to such built-ins, even though c, fail and fail are equivalent, the conjunctions $c,\ X = t$ and $X = t,\ c$ are not. One such built-in is `var/1`: we have, e.g., that `(var(X),X=a)` is not equivalent to `(X=a,var(X))`. Again, independence on the computation 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 (Sahlin 1993, Prestwich 1992) 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 `q(X)` $\leadsto$ `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 backpropagating the bindings generated by `q_u(X,C)` onto `print(X),var(X)` and ensure that a failure of `q_u(X,C)` does not prevent code being generated for `print(X)`. The 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. To link up the solutions of the `findall` with the rest of the unfolding process we use an auxiliary predicate `make_disjunction`. All this leads to the following extra rule, to be added to Definition 16, and where calls whose bindings and whose failure should be hidden are wrapped into a `hide_nf` annotation:

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

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 (in contrast to κ) 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).
```

Let us mark `q(X)` as reducible and wrap it into a `hide_nf()` annotation; the exact representation of the annotated clause required for LOGEN is:

```
ann_clause(1,p(X),(rescall(print(X)),hide_nf(unfold(q(X))))).
```

We now 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).
```

Instead of generating disjunctions, one could also produce new predicates for each disjunction (at least for those cases where argument indexing might be lost (Venken and Demoen 1988)).

4.4 A solution for non-leftmost, non-determinate unfolding

It is well known that non-leftmost, non-determinate unfolding, while sometimes essential for satisfactory propagation of static information, can cause substantial slowdowns. Below we show how our new `hide_nf` annotation can solve this dilemma (another solution is conjunctive partial deduction (Leuschel et al. 1996)).

Example 7
In the following `expensive_predicate(X)` is an expensive, but fully declarative predicate, which for some reason (e.g., termination) we cannot unfold.

```
p(X) :- expensive_predicate(X), q(X), r(X).
q(a).          r(a).
q(b).          r(b).
q(c).
```

If we mark `expensive_predicate(X)` as non-reducible, and `q(X)` and `r(X)` as reducible we get the following residual program:

```
p__0(a) :- expensive_predicate(a).
p__0(b) :- expensive_predicate(b).
```

This residual program has left-propagated the bindings, which is not a problem in itself, but potentially duplicates computations and leads to a less efficient residual program. A solution to this problem, which still allows one to unfold `q(X)` (and right-propagate the bindings onto `r(X)`) and `r(X)` is to wrap them into a `hide_nf` annotation. This is represented as the following annotated clause, where `unfold` is wrapped around calls to be unfolded and `rescall` is wrapped around non-reducible primitives:

```
ann_clause(1,p(X),(rescall(expensive_predicate(X)),
                   hide_nf((unfold(q(X)),unfold(r(X)))))).
```

We then get the following residual program:

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

4.5 Generating correct annotations

Having solved the problem of left-propagation of failure and bindings, we now just have to figure out when `hide_nf` annotations 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.

First, we have to 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 of the generating extension. This relation is defined in Figure 9. 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.

The following modified rule for conjunctions (replacing the corresponding rule in Definition 16) ensures that no bindings are left-propagated or side-effects removed.

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

Here $\mathit{impure}(\kappa_i)$ holds if κ_i contains a call to a side-effect predicate (which has to be non-reducible) or to a non-reducible propagation sensitive call. Calls are classified as in (Sahlin 1993) (e.g., the property of generating a side-effect propagates up the dependency graph). In case we want to prevent backpropagation of bindings on expensive predicates as discussed in Section 4.4, then $\mathit{impure}(\kappa_i)$ should also hold when κ_i contains a call to a non-reducible, expensive predicate.

This modified rule for conjunctions together with Figure 9 can be used to determine the required `hide_nf` annotations. For example, the first rule in Figure 9 actually implies that non-reducible calls never pose a problem and do not have to be wrapped into a `hide_nf` annotation (because they produce the code $\gamma_i = \mathit{true}$ within the generating extension).

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.

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

$$\frac{\forall i: \ \models_{hide} \gamma_i}{\models_{hide} \gamma_1, \ldots, \gamma_n} \qquad \frac{\forall i: \ \models_{hide} \gamma_i}{\models_{hide} (\gamma_1; \gamma_2)}$$

$$\frac{\forall i: \ \models_{hide} \gamma_i}{\models_{hide} (\gamma_1 \rightarrow \gamma_2; \gamma_3)} \qquad \frac{\mathit{hide\_nf}(\kappa) \rightsquigarrow \gamma : \sigma}{\models_{hide} \gamma}$$

Fig. 9. The hide relation $\models_{hide}$

4.6 Negation and Conditionals

Prolog's negation (not/1) is handled similarly 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 the generating extension) cannot fail (otherwise the code generation would be incorrectly prevented) or propagate bindings.

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

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

The first rule is used when we know that κ can be completely and finitely unfolded and it can be determined whether κ 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 we have $\kappa \rightsquigarrow \gamma : \sigma$ with $\sigma \neq \mathit{true}$ then the annotation was wrong and an error will be raised during specialisation. It is thus the responsibility of the BTC to ensure that such errors do not occur.

If the negation is non-reducible then we require that the generating extension does not fail (the hide relation in the premiss). To enable the rule, κ must be given the `hide_nf` annotation unless γ is already hidden. Again, this is the responsibility of the BTC.

Example 8
Consider the following two annotated clauses.

```
p(X)  :-  not(X=a).
q(Y)  :-  not(Y=a).
```

In the first clause `X` is assumed to be of binding-type `static` (or at least `nonvar`) so the negation can be reduced.[9] 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:

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

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

4.7 Disjunctions

To handle disjunctions we will use our `hide_nf` annotation to ensure that failure of one disjunct does not cause the whole specialisation to fail. It will also ensure that the bindings from one disjunct do not propagate over to other disjuncts. The rule for disjunctions therefore has the form:

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

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:

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

The drawback of this rule is 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.

[9] 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.

4.8 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[10] 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 built-in or a 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.
```

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 `=../2` call in clause 2, as non-reducible. Indeed, since the value of `Call` is not known when we generate the unfolding predicate for `map` we should in general not try to 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],(call(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:

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

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),C,Code)` gives `Code = p_u(a,C)`.

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

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

```
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 Experimental Results

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

A first experimental evaluation of the *cogen* approach for Prolog was performed in (Jørgensen and Leuschel 1996). However, due to the limitations of the initial *cogen* only very few realistic examples could be analysed. Indeed, most interesting partial deduction examples require the treatment of partially instantiated data, and the initial *cogen* was thus not very useful in practice. The improved *cogen* of this paper can now deal with such examples and we were able to run our system on a large selection of benchmarks from (Leuschel 1996-2000). We only excluded those benchmarks in (Leuschel 1996-2000) which are specifically tailored towards testing tupling or deforestation capabilities (such as applast, doubleapp, flip, maxlength, remove, rotate-prune, upto-sum, ...), as neither LOGEN nor LIX (nor MIXTUS) will be able to achieve any interesting specialisation on them.

To test the ability to specialise non-declarative built-ins we also devised one new non-declarative benchmark: specialising the non-ground unification algorithm with occurs-check from page 152 of (Sterling and Shapiro 1986) for the query `unify(f(g(a),a,g(a)),S)`. More detailed descriptions about all the benchmarks can be found in (Leuschel 1996-2000).

Our new LOGEN system runs under Sicstus Prolog and is publicly available at `http://www.ecs.soton.ac.uk/~mal` (along with the LIX system). We compare the results of LOGEN with the latest versions of MIXTUS (Sahlin 1993) (version 0.3.6) and ECCE (Leuschel et al. 1998, De Schreye et al. 1999). (Comparisons of the initial *cogen* with other systems such as LOGIMIX, PADDY, and SP can be found in (Jørgensen and Leuschel 1996)). For evaluation purposes, we will also compare with our traditional offline specialiser LIX, which performs exactly the same specialisation as LOGEN (and works on exactly the same annotations). As we have the LOGEN at our disposal, we have not tried to make LIX self-applicable, although we conjecture that, using our

extensions developed in Section 4, it should be feasible to do so (especially since LIX was derived from LOGEN).

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`.

Benchmark	MIXTUS	ECCE		LOGEN		LIX
	with	with	w/o	cogen	genex	
advisor	70 ms	50 ms	20 ms	2.6 ms	0.8 ms	1.7 ms
contains.kmp	210 ms	550 ms	400 ms	1.9 ms	3.6 ms	5.6 ms
ex_depth	200 ms	230 ms	190 ms	1.5 ms	7.2 ms	7.6 ms
grammar	220 ms	200 ms	140 ms	6.3 ms	1.0 ms	1.3 ms
groundunify.simple	50 ms	50 ms	20 ms	6.5 ms	7.7 ms	8.9 ms
groundunify.complex	990 ms	4080 ms	3120 ms	”	8.0 ms	9.3 ms
imperative-solve	450 ms	5050 ms	4240 ms	7.3 ms	4.3 ms	9.2 ms
map.rev	70 ms	60 ms	30 ms	2.7 ms	1.1 ms	1.1 ms
map.reduce	30 ms	60 ms	30 ms	”	1.4 ms	1.4 ms
match.kmp	50 ms	90 ms	40 ms	1.0 ms	2.5 ms	2.8 ms
model_elim	460 ms	240 ms	170 ms	3.0 ms	3.1 ms	3.3 ms
regexp.r1	60 ms	110 ms	80 ms	1.3 ms	1.4 ms	2.0 ms
regexp.r2	240 ms	120 ms	80 ms	”	2.5 ms	4.1 ms
regexp.r3	370 ms	160 ms	120 ms	”	9.9 ms	17.2 ms
ssuply	80 ms	120 ms	60 ms	5.5 ms	0.7 ms	2.4 ms
transpose	290 ms	190 ms	150 ms	1.0 ms	1.9 ms	2.3 ms
ctl	40 ms	160 ms	230 ms	4.4 ms	1.3 ms	1.7 ms
ng_unify	2510 ms	na	na	5.3 ms	3.5 ms	5.7 ms
Total (except ng_unify)	3860 ms	11520 ms	9120 ms	45 ms	58 ms	82 ms
normalised:	66	197	156	0.77	1	1.40

Table 1. *Specialisation Times*

Specialisation Times

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. Note that ECCE can only handle declarative programs, and could therefore not be applied on the *ng_unify* benchmark. 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 for LOGEN and LIX 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. The same

Benchmark	Original	MIXTUS	ECCE	LOGEN / LIX
advisor	1	3.94	3.29	3.94
contains.kmp	1	5.17	6.2	4.89
ex_depth	1	2.16	2.72	2.77
grammar	1	14.40	9.60	15.16
groundunify.simple	1	14.00	14.00	1.56
groundunify.complex	1	14.33	14.33	14.33
imperative-solve	1	1.35	2.56	1.35
map.rev	1	2.30	1.53	1.92
map.reduce	1	3.00	3.60	3.18
match.kmp	1	1.46	1.93	1.15
model_elim	1	3.56	3.78	2.69
regexp.r1	1	6.23	4.26	6.35
regexp.r2	1	2.50	2.57	3.00
regexp.r3	1	3.36	3.14	1.15
ssuply	1	51.00	51.00	51.00
transpose	1	22.71	22.71	22.71
ctl	1	5.85	5.64	5.85
ng_unify	1	4.44	-	3.72
Average Speedup	1	9.25	8.99	8.41
Total Speedup	1	3.63	3.89	2.83

Table 2. *Speedups of the specialised programs*

is true for the LIX column. In general, the binding-type analysis will be the most expensive operation in one-shot applications, and we will address this issue in more detail in the next section. However, the binding-type analysis and the *cogen* have to be run only *once* for every program and division. For example, the generating extension produced for *regexp.r*1 was re-used without modification for *regexp.r*2 and *regexp.r*3 while the one produced for *map.rev* was re-used for *map.reduce*. Another example is the *ctl* interpreter for computation tree logic which is specialised over and over again for different systems and different CTL temporal logic formulas, e.g., in (Leuschel and Lehmann 2000). Hence, in a context where the same program is specialised over and over again for different static values, the time devoted to the *BTA* will usually become negligible.

In summary, the results in this section are valid in a setting where a knowledgeable user can produce a good and safe *BTC* by hand (we have developed a Tcl/Tk based graphical front end that helps the user by providing visual feedback about the annotations) and the same program is re-specialised multiple times.

As can be seen in Table 1, LOGEN and LIX are the fastest specialisation systems overall, running up to almost 3 orders of magnitude faster than the existing online systems. LIX runs roughly 40 % slower than the generating extensions of LOGEN. Note that for 3 benchmarks (*contains.kmp*, *regexp.r*2/3) the cost of running the *cogen* is already re-covered after a single specialisation. All in all, specialisation

times of both LOGEN and LIX are very satisfactory and seem to be more predictable than that of online systems.

Quality of the Specialised Code

Table 2 contains the speedups obtained by the various systems. The table also contains the overall average speedup and total speedup. The latter is a fairer measure than average speedup and is obtained by the formula $\frac{n}{\sum_{i=1}^{n} \frac{spec_i}{orig_i}}$ where n is the number of benchmarks and $spec_i$ and $orig_i$ are the absolute execution times of the specialised and original programs respectively.

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*. *mathitkmp* in Table 2. (To be able to pass the KMP-test, more sophisticated local control would be required, see (Martin and Leuschel 1999) and the discussion below.)

Again, to be fair, both ECCE and MIXTUS are fully automatic systems guaranteeing termination, while for LOGEN sufficient specialisation and termination had to be manually ensured by the user via the *BTC*. 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 shine when compared to the self-applicable SAGE system, where compiler generation usually takes more than 10 hours (with garbage collection) (Gurr 1994) and where the resulting generating extension are still pretty slow (Gurr 1994) (taking more than $100000ms$ to produce the specialised program; unfortunately self-applying SAGE is not possible for normal users and we cannot make exact comparisons with LOGEN).

6 Automating Binding-time Analysis

Automating the process of binding-time analysis has received a lot of attention in the context of functional and imperative languages (Bondorf and Jørgensen 1993, Consel 1993). In the context of logic programs, a major step in achieving automatic binding-time analysis has recently been the use of termination analysis (Bruynooghe, Leuschel and Sagonas 1998, Vanhoof and Bruynooghe 2001). In what follows, we highlight the main aspects of (Vanhoof and Bruynooghe 2001) and report on some experiments.

6.1 Automatic Binding-time Analysis

When annotating a program, one generally wants to mark as many atoms *reducible* as possible, while guaranteeing termination of the unfolding. In order to study the termination characteristics of an unfolding rule $U_{\mathcal{A}}$ associated to an annotation $\mathcal{A}$, we adopt a slightly different notion of annotation from (Vanhoof and Bruynooghe

2001). The basic idea is to represent the annotation $\mathcal{A}$ on a clause by a new clause (which we will call a *t-annotation*) in which the non-reducible atoms are replaced by *true*. This will allow to mimic unfolding using $U_{\mathcal{A}}$ by normal evaluation of the corresponding t-annotation.

Definition 19
Given a clause $H \leftarrow B_1, \ldots, B_n$, a *t-annotated version of* the clause is a clause $H \leftarrow B'_1, \ldots, B'_n$, where for each i such that $1 \leq i \leq n$, it holds that either $B'_i = B_i$ or $B'_i = true$. A t-annotated version of a program $P = \bigcup_i C_i$ is a program $P' = \bigcup_i C'_i$ such that for every such clause C_i, it holds that C'_i is a t-annotated version of C_i.

Note that, according to Definition 19, every clause is a t-annotated version of itself. Given an annotation $\mathcal{A}$ for a program P, we will denote with $\overline{P}_{\mathcal{A}}$ the t-annotated version of P obtained by replacing the atoms that are marked *non-reducible* by $\mathcal{A}$ with *true*. Note that there is a one-to-one correspondence between $\mathcal{A}$, $P_{\mathcal{A}}$ and $\overline{P}_{\mathcal{A}}$ and in what follows we will freely switch between them, referring simply to an "annotated" program. The introduction of a t-annotation allows to reason about the termination behaviour of an unfolding rule $U_{\mathcal{A}}$ when unfolding $P \cup \{G\}$ by studying the termination behaviour of $\overline{P}_{\mathcal{A}}$ with respect to G. Indeed, if $\overline{P}_{\mathcal{A}}$ terminates for a goal G, then the (possibly incomplete) SLD-tree for $P \cup \{G\}$ built by $U_{\mathcal{A}}$ is finite and vice versa.

The above observation is the core of the algorithm developed by (Vanhoof and Bruynooghe 2001), which computes a terminating t-annotation of a program P for a goal G. The basic intuition behind the algorithm, which is depicted in Fig. 10, is as follows: suppose we have to annotate a program P with respect to an initial goal G. If we can prove that G terminates with respect to P, the t-annotated version of P returned by the algorithm is simply P itself (corresponding with a $P_{\mathcal{A}}$ in which every atom is annotated reducible). Hence, $U_{\mathcal{A}}$ constructs a complete SLD-tree for $P \cup \{G\}$ and specialisation of G boils down to plain evaluation. If, on the other hand, termination of G with respect to the t-annotation under construction can not be proven by the analysis due to the presence of a possible loop, the algorithm tries to remove the loop by replacing an atom by *true*. This process is repeated until the constructed t-annotation, and hence the annotated program, is proven to be loop free.

To characterise the possible loops in a program (or a t-annotation) P, the analysis first identifies which of the atoms are *loop-safe*. Intuitively, an atom B_i in a clause $H \leftarrow B_1, \ldots, B_n \in P$ is said to be *loop safe* if the analysis can prove that a finite SLD-tree is built for any atom from the program's callset (the set of calls that can possibly arise during evaluation of $P \cup \{G\}$) that unifies with H if the tree is constructed by unfolding only the i leftmost body atoms of the clause under consideration. Computing whether an atom is loop safe is achieved by known techniques of termination analysis. In our work, we followed the approach of (Codish and Taboch 1999). A norm $\|.\|$ is chosen – mapping a term to a natural number – and the program's callset is approximated by a finite abstract callset, denoted by $calls^{\alpha}_{P}(G)$. Every call in $calls^{\alpha}_{P}(G)$ is of the form $p(b_1, \ldots, b_n)$ with b_i a boolean

stating whether or not the size of that argument (according to the chosen norm) can change upon further instantiation. More formally, we can define the generalisation of a call $p(t_1, \ldots, t_n)$ as $p(\alpha_{\|.\|}(t_1), \ldots, \alpha_{\|.\|}(t_n))$, where $\alpha_{\|.\|}$ is defined as follows, mapping terms onto the boolean domain $\{false, true\}$ with $false > true$:

$$\alpha_{\|.\|}(t) = \begin{cases} true & \text{if } \|t\theta\| = \|t\| \text{ for any } \theta \\ false & \text{otherwise} \end{cases}$$

The abstract callset is kept monovariant – containing a single call per predicate – by taking the predicate-wise least upper bound of the calls in the set. The system then concludes loop-safeness of an atom B_i in a clause $H \leftarrow B_1, \ldots, B_n$ if it can show that there is a guaranteed decrease in size between H and any recursive call that may occur during unfolding of $B_1, \ldots, B_i$ given the calls in $calls^{\alpha}_{P}(G)$ and the size relations between the sizes of the arguments in $B_1, \ldots, B_{i-1}$.

Given the atoms that are guaranteed to be loop safe, the algorithm identifies in each of the clauses the leftmost atom – if it exists – which is not proven to be loop safe, and removes one of these. Note that the algorithm is non deterministic, as

Given a program P and initial goal G.
Let $P_0 = P$, $S_0 = calls^{\alpha}_{P}(G)$, $k = 0$.
repeat
  if there exist a clause i in P_k such that the j'th body atom
  cannot be proven to be loop-safe given S_k
  then
    let P_{k+1} be the program obtained by replacing the j'th
    body atom in the i'th clause in P_k by *true* and
    let $S_{k+1} = S_k \sqcup calls^{\alpha}_{P_{k+1}}(G)$
  else
    $P_{k+1} = P_k$
  $k = k + 1$
until $P_k = P_{k-1}$
$P' = P_k$, $S' = S_k$

Fig. 10. The binding-time analysis algorithm.

several such clauses may exist. Also note the construction of the set S': starting from the program's initial abstract callset S_0, in each round the predicate-wise least upper bound is computed with the current t-annotation's abstract callset. Doing so guarantees that the calls that are unfolded are correctly represented by an abstract call in S', but it also ensures that S' contains abstractions of the (concrete instances of the) calls that were replaced by *true* during the process. In other words, the set S' contains an abstraction of every call that is encountered (unfolded or residualised) during specialisation of P with respect to the initial goal G. Termination of the algorithm is straightforward, since in every iteration an atom in a clause is replaced by *true*, and the program only has a finite number of atoms.

Example 9
Consider the meta interpreter depicted in Fig. 11. The interpreter has the `member/2` and `append/3` predicates as object program.

```
1:solve([]).
2:solve([A|Gs]):- solve_atom(A), solve(Gs).

3:solve_atom(A):-clause(A,Body), solve(Body).

4:clause(member(X,Xs), [append(_,[X|_],_Xs)]).
5:clause(append([],L,L), []).
6:clause(append([X|Xs],Y,[Z|Zs]),[append(Xs,Y,Zs)]).
```

Fig. 11. Vanilla meta interpreter

The binding-time analysis inherits from its underlying termination analysis (Codish and Taboch 1999) the need for a norm to be selected by the user. An often used norm on values of the type $list(T)$ is the so-called *listlength* norm, counting the number of elements in a list. It is defined as follows:

$$\begin{array}{lcl} \| \, [\,] \, \| & = & 0 \\ \| \, [\_ \,|\, Xs] \, \| & = & 1 + \|Xs\| \end{array}$$

Running the binding-time analysis of (Vanhoof and Bruynooghe 2001) on the program depicted in Example 9 with respect to the listlength norm and the initial goal `solve([mem(X,Xs)])` results in an annotated program in which the call to `solve_atom/1` is annotated *non-reducible* and every other call as *reducible*. The resulting abstract callset is

$$\{solve(true), solve\_atom(false), clause(false, false)\}$$

denoting that every call to `solve/1` has an argument that is at least bound to a list skeleton, whereas the arguments in calls to `solve_atom/1` and `clause/2` may be of any instantiation.

Note that there is a close correspondence between the abstract callset and a (monovariant) division. If we define the concretisation function $\gamma_{\|.\|}$ mapping a boolean to a type as $\gamma_{\|.\|}(b) = \tau$ where τ is the most general type such that for all terms $t : \tau$ holds that $\alpha_{\|.\|}(t) \leq b$, then we can define the division corresponding to an abstract callset S as

$$\Delta = \{p(\gamma_{\|.\|}(b_1), ...\gamma_{\|.\|}(b_n)) \mid p(b_1, \ldots, b_n) \in S\}.$$

If $U_{\mathcal{A}}$ and Δ represent, respectively, the unfolding rule and the division corresponding with the t-annotation and abstract callset computed by the binding-time algorithm, then $(U_{\mathcal{A}}, \Delta)$ is a globally safe binding-time classification for the program under consideration.

Example 10
The division corresponding with the callset above is

$$\Delta = \{solve(\texttt{list(dynamic)}), solve\_atom(\texttt{dynamic}), clause(\texttt{dynamic})\}$$

where the parametric type `list(.)` is defined as before:

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

6.2 Additional Experiments

Table 3 summarises a number of experiments that were run with the binding-time analysis of (Vanhoof and Bruynooghe 2001). We could not use all of the benchmarks from Section 5, because the current *BTA* is not yet capable of treating some of the built-ins required and, while it can handle partially static data, it can only handle one kind of partially static data (depending on the single norm with respect to which the program is analysed).

The timings in Table 3 are in milliseconds and were measured on the same machine and Prolog system used in Section 5. The second column (*Round1*) presents the timings for termination analysis of the original program (in which all calls are annotated reducible). In case the outcome of the analysis is possible non-termination, the third column presents the timings for termination analysis of the program from which a call was removed. None of the benchmarks required more than two rounds of the algorithm to derive a terminating t-annotation. The fourth column then contains the total time needed to produce the generating extension using LOGEN and to run it on the partial deduction query. The final column contains the specialisation time of MIXTUS (from Section 5) as a reference point.

Benchmark	Round 1	Round 2	LOGEN	Total	MIXTUS
ex_depth	240.0 ms	230.0 ms	4.4 ms	474 ms	200 ms
match.kmp	470.0 ms	180.0 ms	2.4 ms	652 ms	50 ms
map.rev/reduce	200.0 ms	–	4.3 ms	204 ms	100 ms
regexp.r1-3	740.0 ms	280.0 ms	15.1 ms	1035 ms	670 ms
transpose	210.0 ms	150.0 ms	7.0 ms	367 ms	290 ms
Total	2850 ms		34.5 ms	2885 ms	1330 ms

Table 3. *Timings for the binding-time analysis and full specialisation.*

Note that we slightly modified the *transpose* benchmark in the sense that the first argument is fully static. In fact, in the original *transpose* benchmark the first argument is a list skeleton whose first element in turn is a list skeleton but whose other elements are dynamic. This binding-type cannot be represented precisely by a semi-linear norm (which is required by the termination analysis of (Codish and Taboch 1999) underlying the binding-time analysis).

Analysing Table 3 we can see that the binding-time analysis is indeed the most expensive operation in a one-shot situation. However, the timings are not too bad compared to MIXTUS and the cost of the binding-time analysis will already be recovered after a few specialisations (e.g., after 3 iterations for *ex_depth* and after 2 iterations for *regexp.r3*). Table 4 contains a summary of the speedups obtained by the LOGEN (or LIX) system when using the annotations obtained by the above binding-time analysis. For comparison's sake we have also added the corresponding speedups using the methods of Section 5. Observe that, as was probably to be expected, the automatically generated annotations lead to less speedups than using

Benchmark	Original	MIXTUS	ECCE	LOGEN hand	LOGEN automatic
ex_depth	1	2.16	2.72	2.77	2.23
map.rev	1	2.30	1.53	1.92	1.53
map.reduce	1	3.00	3.60	3.18	1.29
match.kmp	1	1.46	1.93	1.15	1.34
regexp.r1	1	6.23	4.26	6.35	6.35
regexp.r2	1	2.50	2.57	3.00	3.00
regexp.r3	1	3.36	3.14	1.15	1.15
transpose	1	22.71	22.71	22.71	5.89
Average Speedup	1	5.47	5.31	5.28	2.85
Total Speedup	1	2.84	2.85	2.31	1.93

Table 4. *Speedups of the specialised programs*

hand-crafted annotations. Indeed, the hand-crafted annotations for *ex_depth* uses the `hide_nf` annotation to prevent duplication of expensive calls as described in Section 4.4, the hand-crafted annotations for *map.rev* and *map.reduce* uses the special annotations for the `call` primitive described in Section 4.8, while for *transpose* the termination analysis of the automatic *BTA* classified one call as non-terminating which is in fact terminating. Nonetheless, the figures are still pretty good, for the 3 *regexp* benchmarks we obtain exactly the same result as the hand-crafted annotation and for the *match.kmp* the automatic annotation actually outperforms the hand-crafted one.

The conducted experiments show that the approach is feasible and can be automated. However, some issues regarding the current binding-time analysis remain. The analysis basically deals with boolean binding-times: either a value is instantiated enough with respect to a norm, or it is not. Recent research (Genaim, Codish, Gallagher and Lagoon 2002, Vanhoof and Bruynooghe 2002) shows that termination proofs can be constructed by measuring the size of a term by means of a number of simple norms rather than using a single sophisticated norm. These simple norms basically count the number of subterms of the term that are of a particular type. In the presence of type information these norms can be constructed automatically. When combined with information that denotes whether further instantiating a term can introduce more subterms of the particular type they provide a more fine-grained characterisation of a term's size and instantiation. We conjecture such a more detailed characterisation to be a powerful and promising mechanism to derive an automatic binding-time analysis capable of constructing more precise binding-types. Also note that the current analysis only produces monovariant divisions. Polyvariance of the analysis can in principle be obtained by allowing several calls to the same predicate in the abstract callset, creating a new variant of the predicate definition for each abstract call and checking termination of each such predicate separately.

In summary, at least for the experiments in Tables 3 and 4, we can conclude that online systems are to be preferred – both in terms of speed and quality of

the specialised code – in one-shot situations where no expert user is available to perform the annotation. Nonetheless, the quality of the fully automatic LOGEN is satisfactory and the cost of the binding-time analysis will usually be recovered already after a few specialisations. This means that the fully automatic LOGEN might be useful in situations were the same program is specialised multiple times and the specialisation times itself are of utmost importance. Further work is needed to extend and refine the binding-time analysis and to establish its scaling properties for larger programs.

7 Discussion and Future Work

7.1 Related Work

The first hand-written compiler generator based on partial evaluation principles was, in all probability, the system *RedCompile* (Beckman, Haraldson, Oskarsson and Sandewall 1976) for a dialect of Lisp. Since then successful compiler generators have been written for many different languages and language paradigms (Romanenko 1988, Holst 1989, Holst and Launchbury 1991, Birkedal and Welinder 1994, Andersen 1994, Glück and Jørgensen 1995, Thiemann 1996).

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 (Neumann 1990, Neumann 1991). However, it is not based on (off-line) partial deduction.

Also the construction of our program $P_u^{\mathcal{A}}$ (Definition 16) is related to the idea of *abstract compilation* (Hermenegildo, Warren and Debray 1992, Codish and Demoen 1995). In abstract compilation a program P is first transformed and abstracted. Evaluation of this transformed program corresponds to the actual abstract interpretation analysis of P. 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 (Tarau and De Bosschere 1994) to calculate abstract answers. Finally, (Gallagher and Lafave 1996) uses a source-to-source 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).

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 (Martin and Leuschel 1999). 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, (Martin and Leuschel 1999) only addresses the local control component and it is still unclear how it can be extended for the global control (the prototype in

(Martin and Leuschel 1999) uses the online ECCE system for global control; to this end trace terms were built up in the generating extension like in (Gallagher and Lafave 1996)).

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 analogous 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 (Launchbury 1991) and essentially, our system performs specialisation that a partial evaluation system for a functional language would need some form of *driving* (Glück and Sørensen 1994) to be able to do. However, our divisions are still congruent: the value of a static variable cannot depend on a dynamic value. In the above example, the value of X within the call $q(X)$, if reached, is always going to be a, no matter what the argument to `g` is.

7.2 Mixline Specialisation

Some built-ins 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 mixline (Jones et al. 1993) treatment of some built-ins.

$$\underline{c} \rightsquigarrow c : c \qquad \text{if } c = \texttt{var}(t), \texttt{copy\_term}(s,t), s\backslash\texttt{==}t, \ldots$$
$$\underline{c} \rightsquigarrow (c \rightarrow C = \mathit{true}; C = c) : C \qquad \text{if } c = \texttt{ground}(t), \texttt{nonvar}(t), \texttt{atom}(t), \texttt{integer}(t), s\texttt{==}t, \ldots$$

The code of the *cogen* in Appendix A uses these optimisations if a `mixcall` annotation is used (these annotations have not been used for the experimental results in Section 5). It also contains a mixline 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 `mix`, which lies in between `static` and `dynamic` (Jones et al. 1993). Basically, `mix` behaves like `static` for the generalisation gen_Δ (Definition 14) but like `dynamic` for filtering $filter_\Delta$ (Definition 15). The former means that an argument marked as `mix` 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 *mixline* 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 can be achieved, without having to change the *cogen* itself, by modifying the annotation process. Indeed, instead of marking a call $p(t_1, \ldots, t_n)$ either as reducible or non-reducible we simply insert a static conditional into the annotated program: `(`*Test* `->` $\underline{p}(t_1, \ldots, t_n)$ `;` $p(t_1, \ldots, t_n)$`)`. 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 mixline 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 (Sørensen and Glück 1999)).

7.3 More Future Work

In addition to extending our *BTA* to generate `hide_nf` annotations and to fully integrate the *BTA* into the LOGEN system, one might also think of further 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 (Glück and Jørgensen 1995). 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.

Another interesting recent development is fragmental specialisation (Helsen and

Thiemann 2000), where the idea is to specialise fragments of the code (such as modules) in the order in which they arrive. It should be possible to add such a capability to our *cogen*, by using co-routining features (e.g., of SICStus Prolog) so as to suspend, for predicates p defined in other fragments, calls to the corresponding p_u or p_m predicates until the fragment defining p is available.

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. Further work is needed to establish whether it is possible to adapt the *cogen* approach for Gödel (Hill and Lloyd 1994) or Mercury (Somogyi et al. 1996) so that it still produces efficient generating extensions.

Finally, it also seems natural to investigate to what extent more powerful control techniques (such as characteristic trees (Gallagher and Bruynooghe 1991, Leuschel et al. 1998), trace terms (Gallagher and Lafave 1996) or the local control of (Martin and Leuschel 1999)) and specialisation techniques (like conjunctive partial deduction (Leuschel et al. 1996, Glück, Jørgensen, Martens and Sørensen 1996, De Schreye et al. 1999)) can be incorporated into the *cogen*, while keeping its advantages in terms of efficiency.

7.4 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 procedure 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 data structures, declarative and non-declarative built-ins, disjunctions, conditionals, and negation. We have shown that the resulting system achieves fast specialisation in situations where the same program is re-specialised multiple times. We have also overcome several limitations of earlier offline systems and shown that LOGEN can be applied on a wide range of natural logic programs and that the resulting specialisation is also very good, sometimes even surpassing that of existing online systems. We have also developed the foundation for a fully automatic binding-type analysis for the LOGEN system, and have evaluated its performance on several examples.

Acknowledgements

We thank Michael Codish, Bart Demoen, Danny De Schreye, André de Waal, Robert Glück, Gerda Janssens, Neil Jones, Bern Martens, Torben Mogensen, Ulrich Neumerkel, Kostis Sagonas, and Peter Thiemann for interesting discussions and contributions on this work. My thanks also go to Laksono Adhianto for developing the graphical user interface. Finally, we are very grateful to anonymous referees for their helpful comments and constructive criticism.

References

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

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

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

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

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

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

Bondorf, A. and Jørgensen, J. (1993). Efficient analyses for realistic off-line partial evaluation, *Journal of Functional Programming* **3**(3): 315–346.

Bondorf, A., Frauendorf, F. and Richter, M. (1990). An experiment in automatic self-applicable partial evaluation of Prolog, *Technical Report 335*, Lehrstuhl Informatik V, University of Dortmund.

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

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

Chen, W., Kifer, M. and Warren, D. S. (1989). A first-order semantics of higher-order logic programming constructs, *in* E. L. Lusk and R. A. Overbeek (eds), *Logic Programming: Proceedings of the North American Conference*, MIT Press, pp. 1090–1114.

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

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

Consel, C. (1993). Polyvariant binding-time analysis for applicative languages, *PEPM93*, ACM, pp. 66–77.

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

Cosmadopoulos, Y., Sergot, M. and Southwick, R. W. (1991). Data-driven transformation of meta-interpreters: A sketch, *in* H. Boley and M. M. Richter (eds), *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, Vol. 567 of *LNAI*, Springer Verlag, Kaiserslautern, FRG, pp. 301–308.

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

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

Futamura, Y. (1971). Partial evaluation of a computation process — an approach to a compiler-compiler, *Systems, Computers, Controls* **2**(5): 45–50.

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

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

Gallagher, J. and Bruynooghe, M. (1990). Some low-level transformations for logic programs, *in* M. Bruynooghe (ed.), *Proceedings of Meta90 Workshop on Meta Programming in Logic*, Leuven, Belgium, pp. 229–244.

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

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

Genaim, S., Codish, M., Gallagher, J. and Lagoon, V. (2002). Combining norms to prove termination, *in* T. Cortesi (ed.), *Verification, Model Checking and Abstract Interpretation, VMCAI2002, Revised Papers*, LNCS 2294, Springer-Verlag, pp. 126-138.

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

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

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

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

Helsen, S. and Thiemann, P. (2000). Fragmental specialization, *in* W. Taha (ed.), *Proceedings of SAIG'00*, LNCS 1924, Springer-Verlag, pp. 51–71.

Hermenegildo, M., Warren, R. and Debray, S. K. (1992). Global flow analysis as a practical compilation tool, *The Journal of Logic Programming* **13**(4): 349–366.

Hill, P. and Gallagher, J. (1998). Meta-programming in logic programming, *in* D. M. Gabbay, C. J. Hogger and J. A. Robinson (eds), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 5, Oxford Science Publications, Oxford University Press, pp. 421–497.

Hill, P. and Lloyd, J. W. (1994). *The Gödel Programming Language*, MIT Press.

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

Holst, C. K. and Launchbury, J. (1991). Handwriting cogen to avoid problems with static typing, *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, Glasgow University, pp. 210–218.

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

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

Jørgensen, J. and Leuschel, M. (1996). Efficiently generating efficient generating exten-

sions in Prolog, *in* O. Danvy, R. Glück and P. Thiemann (eds), *Partial Evaluation, International Seminar*, LNCS 1110, Springer-Verlag, Schloß Dagstuhl, pp. 238–262.

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

Launchbury, J. (1991). *Projection Factorisations in Partial Evaluation*, Distinguished Dissertations in Computer Science, Cambridge University Press.

Leuschel, M. (1996-2000). The ECCE partial deduction system and the DPPD library of benchmarks, Obtainable via `http://www.ecs.soton.ac.uk/~mal`.

Leuschel, M. and Bruynooghe, M. (2002). Logic program specialisation through partial deduction: Control issues, *Theory and Practice of Logic Programming* **2**(4 & 5): 461–515.

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

Leuschel, M. and Lehmann, H. (2000). Solving coverability problems of Petri nets by partial deduction, *in* M. Gabbrielli and F. Pfenning (eds), *Proceedings of PPDP'2000*, ACM Press, Montreal, Canada, pp. 268–279.

Leuschel, M. and Sørensen, M. H. (1996). Redundant argument filtering of logic programs, *in* J. Gallagher (ed.), *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, Springer-Verlag, Stockholm, Sweden, pp. 83–103.

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

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

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

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

Martens, B. and De Schreye, D. (1995). Two semantics for definite meta-programs, using the non-ground representation, *in* K. R. Apt and F. Turini (eds), *Meta-logics and Logic Programming*, MIT Press, pp. 57–82.

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

Martens, B. and Gallagher, J. (1995). Ensuring global termination of partial deduction while allowing flexible polyvariance, *in* L. Sterling (ed.), *Proceedings ICLP'95*, MIT Press, Kanagawa, Japan, pp. 597–613.

Martin, J. and Leuschel, M. (1999). Sonic partial deduction, *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, Springer-Verlag, Novosibirsk, Russia, pp. 101–112.

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

Neumann, G. (1990). Transforming interpreters into compilers by goal classification, *in* M. Bruynooghe (ed.), *Proceedings of Meta90 Workshop on Meta Programming in Logic*, Leuven, Belgium, pp. 205–217.

Neumann, G. (1991). A simple transformation from Prolog-written metalevel interpreters into compilers and its implementation, *in* A. Voronkov (ed.), Logic Programming. *Pro-*

ceedings of the First and Second Russian Conference on Logic Programming, LNCS 592, Springer-Verlag, pp. 349–360.

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

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

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

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

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

Sørensen, M. H. and Glück, R. (1995). An algorithm of generalization in positive supercompilation, *in* J. W. Lloyd (ed.), *Proceedings of ILPS'95, the International Logic Programming Symposium*, MIT Press, Portland, USA, pp. 465–479.

Sørensen, M. H. and Glück, R. (1999). Introduction to supercompilation, *in* J. Hatcliff, T. Æ. Mogensen and P. Thiemann (eds), *Partial Evaluation — Practice and Theory*, LNCS 1706, Springer-Verlag, Copenhagen, Denmark, pp. 246–270.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*, MIT Press.

Tarau, P. and De Bosschere, K. (1994). Memoing techniques for logic programs, *in* Y. Deville (ed.), Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'93*, Workshops in Computing, Springer-Verlag, Louvain-La-Neuve, Belgium, pp. 196–209.

Thiemann, P. (1996). Cogen in six lines, *International Conference on Functional Programming*, ACM Press, pp. 180–189.

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

Vanhoof, W. (2000). Binding-time analysis by constraint solving: a modular and higher-order approach for mercury, *in* M. Parigot and A. Voronkov (eds), *Proceedings of LPAR'2000*, LNAI 1955, Springer-Verlag, pp. 399–416.

Vanhoof, W. and Bruynooghe, M. (2001). Binding-time annotations without binding-time analysis, *in* R. Nieuwenhuis and A. Voronkov (eds), *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, Proceedings*, Vol. 2250 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 707–722.

Vanhoof, W. and Bruynooghe, M. (2002). When size does matter - Termination analysis for typed logic programs, *in* A. Pettorossi (ed.), *Logic-based Program Synthesis and Transformation, Proceedings of LOPSTR 2001*, Vol. 2372 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 129–147.

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

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

Yardeni, E., Frühwirth, T. and Shapiro, E. (1992). Polymorphically typed logic programs, *in* F. Pfenning (ed.), *Types in Logic Programming*, MIT Press, pp. 63–90.

A The Prolog cogen

This appendix contains the listing of the cogen. It works on an annotated version of the program to be specialised which contains definitions for the following predicates:

- `residual`: defines the predicates by which the generating extension is to be called, as well as the predicates which are residualised.
- `filter`: the division for the residual predicates
- `ann_clause`: the annotated clauses where calls in the body are annotated by:
  - `unfold` for reducible user-defined predicates, and `memo` for non-reducible user-defined predicates,
  - `call` for reducible primitives (i.e., built-ins or open predicates; c.f., Section 4), and `rescall` for non-reducible user-defined predicates,
  - `semicall` for non-reducible primitives to be specialised in a mixline fashion (c.f., Section 7.2),
  - `ucall` for a `call` primitive calling a reducible user-defined predicate and `mcall` for a `call` primitive calling a non-reducible user-defined predicate (c.f., Section 4.8),
  - `if` and `resif` for reducible and non-reducible conditionals respectively, and `semif` for conditionals to be specialised in a mixline fashion (c.f., Section 7.2),
  - `not` and `resnot` for reducible and non-reducible negations respectively,
  - `;` and `resdisj` for reducible and non-reducible disjunctions respectively,
  - `hide`, `hide_nf` to prevent the propagation of bindings and failure.

An example annotated file can be found in Appendix B.

```
/* ----------- */
/*  C O G E N  */
/* ----------- */
:- ensure_consulted('pp').

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

memo_clause(clause(Head,(find_pattern(Call,V) ->
                         true ;
                         (insert_pattern(GCall,Hd),
                          findall(NClause,
                                  (RCall, NClause = clause(Hd,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,Hd),
                          findall(NClause,
```

```
                                    (RCall, NClause = clause(Hd,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))))
           )
     )), SpecCode) :-
 /* RG1,RG2,RG3 shouldn't fail and be determinate */
        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),\+(RG1),true) :- body(G1,RG1,_VS1).
body(resnot(G1),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).

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(mix,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))).
```

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(1,nont(X,T,R), (unfold(t(a,T,V)),memo(nont(X,V,R)))).
ann_clause(2,nont(X,T,R), (unfold(t(X,T,R)))).
ann_clause(3,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 not 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).
```

The generating extension is usually executed using `nont_m`, whereby the last argument is instantated to the filtered version of the call under consideration. E.g., to specialise the original program for `nont(c,T,R)` we call `nont_m(c,T,R,FCall)`, which instantiates `FCall` to `nont__0(T,R)` and prints the following residual program:

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

Observe that we can use the computed answer substitution for `FCall` to produce an interface definition clause:

```
nont(c,T,R) :- nont__0(T,R).
```

This will be done automatically by the LOGEN system when it produces the specialised program.

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 (Mogensen and Bondorf 1992) (where we obtain deterministic automaton after specialisation) can be found in (Jørgensen and Leuschel 1996).

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(1,transpose(A,[]),unfold(nullrows(A))).
ann_clause(2,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(3,makerow([],[],[]),true).
ann_clause(4,makerow([[A|B]|C],[A|D],[B|E]),unfold(makerow(C,D,E))).
filter(nullrows(A),[type(list(type(list(dynamic))))]).
ann_clause(5,nullrows([]),true).
ann_clause(6,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 for `transpose([[a,b],[c,d]],R)` leads to the following specialised program (and full unfolding has been achieved):

```
transpose([[a,b],[c,d]],A) :- transpose__0(a,b,c,d,A).
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)].
```

Using this division, the specialised program for `transpose([[a,b],[c,d]],R)` is:

```
transpose([[a,b],[c,d]],A) :- transpose__0(a,b,[c,d],A).
transpose__0(B,C,[D,E],[[B,D],[C,E]]).
```