

Collecting Potential Optimisations

*Nancy Mazur
Gerda Janssens
Wim Vanhoof*

Report CW357, February 2003



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Collecting Potential Optimisations

Nancy Mazur
Gerda Janssens
Wim Vanhoof

Report CW 357, February 2003

Department of Computer Science, K.U.Leuven

Abstract

In this paper we describe an analysis system for logic programs that makes it possible to characterise the optimisation opportunities within a predicate. The idea behind the proposed framework is that for each predicate the literals that can potentially be optimised are identified and collected. The conditions under which these optimisations can be done safely are expressed as requirements on the calls to the predicate and collected into a so-called (local) optimisation table. This table does not only give a view on the potential of optimisation in a program, but as it relates the optimisations with call substitutions, it also defines the circumstances in which an optimisation can occur. This information can be valuable input during the development of better version generation heuristics, or can be given as feedback to the programmer.

Keywords : logic programming, program analysis, compiler optimisations, polyvariance, version control

CR Subject Classification : D.1.7, D.3.4, F.3.2

Collecting Potential Optimisations

Nancy Mazur, Gerda Janssens, and Wim Vanhoof

Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{nancy,gerda,wimvh}@cs.kuleuven.ac.be

Abstract. In this paper we describe an analysis system for logic programs that makes it possible to characterise the optimisation opportunities within a predicate. The idea behind the proposed framework is that for each predicate the literals that can potentially be optimised are identified and collected. The conditions under which these optimisations can be done safely are expressed as requirements on the calls to the predicate and collected into a so-called (local) optimisation table. This table does not only give a view on the potential of optimisation in a program, but as it relates the optimisations with call substitutions, it also defines the circumstances in which an optimisation can occur. This information can be valuable input during the development of better version generation heuristics, or can be given as feedback to the programmer.

1 Introduction

A recurring problem which most optimising compilers or program transformers face is the problem of generating the adequate versions for each program entity that can be optimised in order to obtain a sufficiently optimised program [18, 19, 25, 14, 11, 24]. In a logic programming setting, the entities to be optimised are for example predicates. If each predicate can be optimised in a number of different ways, it is important to know which versions are interesting. If too many versions are generated, the size of the transformed program may become too large with respect to the efficiency gained by the optimisations that these versions allow. On the other hand, if the version generation policy is too restrictive, the program may be suboptimal, and still not as efficient as one might have hoped.

A classic way of guiding the version generation process during the transformation of a program is to use a top-down program analysis system which is able to detect the different uses of a predicate [24, 25, 14]. For each use of a predicate, an adequate optimised version can be generated. Usually, this top-down approach still generates too many versions, therefore a pruning phase may follow [24]. The disadvantage of these approaches is that for each new use of a predicate, the program needs to be reanalysed as the underlying analyses can not guarantee that the optimisations are safe for each new use of the predicate. A typical example of program optimisation is the compile-time garbage-collection (CTGC) system we developed previously [19]. In this system we used a different approach. For each predicate we generate at most two versions: a plain non-optimised version

that is always safe to call, and a fully optimised version that can only be used if the caller of the predicate meets the (harsh) conditions in order to guarantee that all the optimisations are safe. Obviously, there is no risk for code explosion, yet a lot of intermediate opportunities for optimisations are missed.

The common problem in the above approaches is that, although we are able to spot the optimisations, we do not have an adequate mechanism for collecting and comparing the possible optimisations *before* the actual versions are created. Therefore, we have developed an *optimisation derivation system* which is actually an analysis tool that is capable of spotting possible optimisations and relating these optimisations to the calls for which they are safe. Indeed, if we can collect the set of possible optimisations within a predicate and relate each optimisation to a requirement on the calls of that predicate, then we can (automatically) reason about the versions that are interesting to generate. For example, if more than one optimisation is spotted for the same call description, then this may be a reason for generating a version with the corresponding optimisations; if a predicate has in its body a recursive call and the recursive call allows only one type of optimisation, then it might not be worthwhile to consider any other optimisations in this predicate. The relation between possible optimisations and a description of the calls that allow them is also interesting feedback to the programmer: Why can an atom at some program point not be optimised? How does a predicate need to be called in order to profit from the full optimisation potential? It can also be seen as a step towards conceptually separating the analysis of a predicate from the version generation which depends on it, opening the field for better insights into the latter problem.

We assume that the input to our optimisation derivation system is a program in which each program point is already annotated with elements from some basic abstract domain upon which the intended optimisations can be decided. In the context of CTGC, where the intended memory optimisations depend on liveness analysis which itself depends on possible alias¹ and in use information, we expect that this underlying information is already at hand before reasoning about the optimisation opportunities. We assume that these annotations are call independent², and that they can be used to approximate call dependent situations using the appropriate combination operator. This is not too much of a restriction, as we want to continue to be able to support modular programming, which requires a call independent approach to program analysis anyway.

Given a specification of the intended type of optimisation and a correctly annotated program, our analysis is able to identify all the literals within the program that can *potentially* be optimised with respect to this specification. This means that our optimisation derivation system intrinsically overestimates the optimisation opportunities. As a consequence extra controls are needed when

¹ This is the domain of structure sharing [21, 20], i.e. a domain which captures the information of whether two terms share some structure on the heap or not. This domain is different from the set-sharing domain [4, 2] which tracks which variables of partially instantiated data-structures are shared.

² This is the case for the domains used in CTGC, including the possible alias domain.

using the results for generating the appropriate safe versions. It is possible to avoid this overestimation, but then we would need to decide which optimisations to keep, which is equivalent to the original versioning problem.

The main contribution of this paper is a framework that embodies our novel approach to the characterisation of the optimisation opportunities within a predicate. We identify each optimisable literal by a description of the calls for which the optimisation is safe. Therefore, given a specific call substitution it becomes straightforward to verify what optimisations it allows.

Section 2 specifies the logic programming language for which the proposed analysis system is elaborated, recalls some basics about program analysis and introduces some of the concepts used later on. Using a straightforward optimisation as working example, we describe the intuition of our approach in Section 3. This is formalised in Section 4. In Section 5, we propose an improvement by considering sets of optimisation opportunities instead of single optimisation descriptions. The overall system is discussed in Section 6 and related to existing work in Section 7. Section 8 concludes the work.

2 Preliminaries

The target language is a logic programming language consisting of definite clauses, extended with explicit disjunctions, if-then-else constructs and negation. Let \mathcal{V} denote the set of program variables, Σ the set of functor symbols, and Π the set of predicate symbols. We use \overline{X} as notation for a sequence X_1, \dots, X_n .

$$\begin{aligned}
 \textit{Program} & ::= p_1 \dots p_{n_p} & n_p \geq 1 \\
 \textit{Procedure} & ::= q(\overline{X}_1, \dots, \overline{X}_n) \leftarrow g \\
 \textit{Goal} & ::= g_1, g_2 \mid g_1; g_2 \mid \textit{if } g_1 \textit{ then } g_2 \textit{ else } g_3 \mid \textit{not } g \mid l \\
 \textit{Literal} & ::= X = Y \mid X = f(Y_1, \dots, Y_m) \mid q(\overline{X}_1, \dots, \overline{X}_n)
 \end{aligned}$$

where

$$\begin{aligned}
 \{p_1, \dots, p_{n_p}\} & \subseteq \textit{Procedure} \\
 \{g, g_1, g_2, g_3\} & \subseteq \textit{Goal} \\
 l & \in \textit{Literal} \\
 q & \in \Pi \\
 \{X, Y, X_1, \dots, X_n, Y_1, \dots, Y_m\} & \subseteq \mathcal{V} \\
 f & \in \Sigma
 \end{aligned}$$

Fig. 1. Syntax of our first order logic programming language.

Although most modern logic programming languages allow the use of modules [10], for the time being we consider that a program is defined as a monolithic block consisting of a number of predicates. Figure 1 defines the syntax of our

language in which each predicate is described by exactly one procedure. A procedure consists of a head atom and a goal. Goals can be either conjunctions or disjunctions of subgoals, if-then-else constructs, negations or simple literals. Literals are either explicit unifications, namely $X = Y$ or $X = f(Y_1, \dots, Y_m)$, or calls to procedures $p(X_1, \dots, X_n)$ (which can be either calls to builtin procedures, or to procedures defined in the program). All variables occurring in a head atom or a literal are distinct. This syntax corresponds to a so-called “normal form” representation. Each logic program that is defined in terms of predicates with several clauses and literals containing general terms instead of distinct variables can straightforwardly be translated into normal form by introducing explicit disjunctions, breaking up complicated unifications, and appropriately renaming the variables.

Each literal in a program is implicitly identified by a *program point*. The set of program points within a program is denoted by pp . We use the notation l_i to designate the literal at program point $i \in \text{pp}$. We use $\text{pp}(l)$ to denote the program point of a particular literal $l \in \text{Literal}$. Likewise, the set of program points occurring in the definition of a procedure $p \in \text{Procedure}$ is denoted as $\text{pp}(p)$. If we explicitly write a program point within a code fragment, we do this by writing it (as a natural number) in front of the literal to which it refers. This is in accordance with the fact that we are interested in *how* a literal is called, which is important for deciding whether or not an optimisation is possible.

We use the top-down collecting semantics [5] for describing the concrete semantics of logic programs and consider that this semantics is expressed in terms of elements from a domain \mathcal{C} . \mathcal{C} is required to be a complete lattice: $\langle \mathcal{C}, \subseteq, \cup, \cap, \top_{\mathcal{C}}, \perp_{\mathcal{C}} \rangle$. The abstract counterpart of \mathcal{C} is a domain \mathcal{A} , also a complete lattice $\langle \mathcal{A}, \sqsubseteq, \sqcup, \sqcap, \top_{\mathcal{A}}, \perp_{\mathcal{A}} \rangle$. To ensure a terminating fixpoint computation, we require this domain to be Noetherian³. Both domains capture the information which is necessary for detecting the intended optimisations. While the concrete domain is used to capture the relevant parts of the program environments as they occur while executing the program, the latter domain, \mathcal{A} is used to approximate these environments at compile time. In most cases this *relevant part* describes the variable bindings as they occur at each stage of the program, limited to the variables occurring in the context of the procedure that is being executed. The concrete domain that is typically used to express this information is the domain of idempotent variable substitutions $\langle \wp(\text{ESubst}), \subseteq, \cup, \cap, \wp(\text{ESubst}), \emptyset \rangle$ [15, 6]. But given the fact that other elements of the run time environment can be of interest for the concrete domain too, we do not limit ourselves to this domain.

The concrete and abstract domains are connected by a so-called *concretisation* function $\gamma : \mathcal{A} \rightarrow \mathcal{C}$. This function is monotone, strict (i.e. $\gamma \perp_{\mathcal{A}} = \perp_{\mathcal{C}}$), and co-strict (i.e. $\gamma \top_{\mathcal{A}} = \top_{\mathcal{C}}$). Specific elements from \mathcal{C} or \mathcal{A} are called *concrete substitutions* and *abstract substitutions* resp., or *substitutions* in general. Substitutions are called *call substitutions* if they describe the calling environment of a procedure call. They are called *exit substitutions* if they describe the environ-

³ Noetherian domains are domains in which all ascending chains have finite length [17].

ment at procedure exit⁴. In the examples that refer to $\wp(ESubst)$ as the concrete domain, we call an element from $\wp(ESubst)$ a *set of variable substitutions* instead of a concrete substitution.

Let $\delta \in \mathcal{A}$, and $\sigma \in \mathcal{C}$, then δ is said to be a *safe approximation* of σ , denoted by $\delta \propto \sigma$, iff $\sigma \subseteq \gamma(\delta)$.

Example 1. A classic application of program analysis in logic programming is groundness analysis [16, 9, 12]. A typical abstract domain used to represent groundness information is the domain of positive boolean expressions Pos [16], extended with the bottom element `false`: $\langle Pos, \sqsubseteq, \vee, \wedge, \text{true}, \text{false} \rangle$. The concretisation of abstract substitutions in Pos is usually expressed with respect to the concrete domain of variable substitutions: $\langle \wp(ESubst), \subseteq, \cup, \cap, \wp(ESubst), \emptyset \rangle$. A variable substitution $\theta \in ESubst$ for some variables X and Y can be $\theta = \{X/a, Y/f(X)\}$, which means that X is bound to a constant a , and Y is bound to a term with top-level functor $f/1$ and argument X .

The concretisation function γ is defined as the function mapping an element $\delta \in Pos$ to a set of variable substitutions $\sigma \in \wp(ESubst)$ such that each variable in each variable substitution $\theta \in \sigma$ is at least as ground as its groundness described in δ . We have $\gamma \text{false} = \emptyset$, and $\gamma \text{true} = \wp(ESubst)$.

Let $\sigma = \{\{Y/b\}, \{Y/f(X)\}\}$, and $\delta_1, \delta_2 \in Pos$ where $\delta_1 = y$, $\delta_2 = x \rightarrow y$, then $\delta_1 \not\propto \sigma$, $\delta_2 \propto \sigma$.

In the concrete domain, we use the notation $p \overset{\sigma}{\rightsquigarrow} i$, with $p \in Procedure$, $\sigma \in \mathcal{C}$, $i \in \text{pp}(p)$, to denote the concrete substitution that describes the relevant part of the runtime environments (indeed, it can be more than one in the presence of backtracking) at program point i , i.e. *before* executing the literal at that program point, within a procedure p when p is called with the substitution σ . In this definition, the particular selection rule used in the language is implicitly taken into account: changing the selection rule means that the value of $p \overset{\sigma}{\rightsquigarrow} i$ changes too. We say that $p \overset{\sigma}{\rightsquigarrow} i$ is the concrete substitution *induced* by σ at program point i in p .

Example 2. Let the predicate $p/1$ be defined by the following procedure (with `append(X, Y, Z)` defined as the usual concatenation of lists of elements, see page 7):

`p(X) :-` ₁ `C = [1],` ₂ `append(A, B, C),` ₃ `X = A.`

where 1, 2 and 3 are the program points referring to the literals `C = [1]`, `append(A, B, [1])` and `X = A` resp.

For the call substitution $\sigma = \{\{\}\} \in \wp(ESubst)$, we have

$$p \overset{\sigma}{\rightsquigarrow} 3 = \{\{A/[], B/[1], C/[1]\}, \{A/[1], B/[], C/[1]\}\}.$$

In the introduction we mentioned that our programs are pre-annotated with all the information that is needed for our optimisation derivation system. To formalise this idea we introduce the notion of goal-independent annotations and their safeness with respect to the concrete domain.

⁴ We assume that call substitutions as well as exit substitutions of a literal $p(X_1, \dots, X_n)$ are restricted to the variables $\{X_1, \dots, X_n\}$.

Definition 1 (Goal-Independent Annotations). A goal-independent annotation table is a function mapping individual program points to abstract substitutions, $Ann : pp \rightarrow \mathcal{A}$. The abstract substitutions recorded in such a table are denoted by the letter ι . We refer to these substitutions as the goal-independent annotations.

These annotations are typically the result of a goal-independent program analysis. The meaning of these annotations w.r.t. the concrete domain is defined by the notion of safeness which is defined as follows:

Definition 2 (Safeness). A goal-independent annotation $\iota_i \in \mathcal{A}$ of a program point i belonging to a procedure p is safe w.r.t. a concrete domain \mathcal{C} and an abstract combination operator $\otimes : \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{A}$ iff for each call substitution $\delta \in \mathcal{A}$ of the procedure p we have that if $\sigma \in \mathcal{C}$ is a concrete substitution such that $\delta \propto \sigma$, then $(\iota_i \otimes \delta) \propto (p \stackrel{\sigma}{\rightarrow} i)$.

A goal-independent annotation table $A : pp \rightarrow \mathcal{A}$ is said to be safe w.r.t. a concrete domain \mathcal{C} and a combination operator $\otimes : \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{A}$ iff each goal-independent annotation ι within this annotation table is safe w.r.t. \mathcal{C} and \otimes .

The above definition is given in terms of an *abstract combination operator*. In most domains, this combination operator is \sqcap (see also the following example), but in general, other operators can be used. A more thorough discussion is presented in Section 6.

Example 3. Using *Pos* as the domain for representing groundness information, $\{(1, \iota_1), (2, \iota_2), (3, \iota_3)\}$ where $\iota_1 = \text{true}$, $\iota_2 = c$, and $\iota_3 = a \wedge b \wedge c$, is a safe goal-independent annotation table for $p/1$ (Example 2) with respect to the concrete domain $\wp(ESubst)$ and the abstract operation \wedge , i.e. the greatest lower bound. Indeed, it suffices to conjoin a particular goal-independent annotation with an abstract call substitution in order to obtain a safe description of the concrete substitutions that are induced by the concrete call substitutions approximated by that abstract call substitution. Consider for example a call of $p(X)$ in which X is ground. This can be described by the abstract substitution δ which is equal to the *Pos*-formula x . In that case, at runtime we have that at program point 2, before the call to `append`, X as well as C are always ground. This is safely described by the abstract substitution $\iota_2 \wedge \delta$ which is equal to $c \wedge x$.

The collection of substitutions, terms, atoms or any other syntactic elements of our language are called *objects*. The set of variables occurring in an object x is denoted by $Vars(x)$. A *renaming* $\rho_{\overline{X} \rightarrow \overline{Y}}$ is an idempotent function which, when applied to an object x produces a new object in which all occurrences of the variables X_1, \dots, X_n have been replaced by the corresponding variables Y_1, \dots, Y_n . We assume that $\{X_1, \dots, X_n\} \cap \{Y_1, \dots, Y_n\} = \emptyset$ and denote the application of a renaming $\rho_{\overline{X} \rightarrow \overline{Y}}$ to x as $\rho_{\overline{X} \rightarrow \overline{Y}}x$. Given two objects x and y , if a renaming ρ exists such that $\rho x = y$ (which also means that there exists a renaming ρ' such that $\rho' y = x$), then x and y are called variants with respect

to each other. This is denoted as $x \approx y$. We use the notation $x_{/\approx}$ to represent the set of objects which are variants of x , x is then used as a representative of this set. The *projection* of a substitution δ on a set of variables V , denoted by $\delta|_V$, is a new substitution obtained by universal projection of δ on this set of variables. Let $\bar{V} = \text{Vars}(\delta) \setminus V$, we have $\delta|_V = (\forall (X \in \bar{V}). \delta)$. This operation eliminates the variables not occurring in V from the substitution: $\text{Vars}(\delta|_V) \subseteq V$. Note that $\delta|_V \sqsubseteq \delta$, for all substitutions δ . Both operations, renaming applied on substitutions as well as projection on a set V , are monotone.

Example 4. Let δ be equal to $x \wedge y \in \text{Pos}$. Using $[]$ to denote sequences, let $S_1 = [x, y]$, $S_2 = [a, b]$, and $V = \{x\}$, then $\rho_{S_1 \rightarrow S_2} \delta = a \wedge b$ and $\delta|_V = \forall v \in (\text{Vars}(\delta) \setminus V). \delta = \forall v \in \{y\}. \delta = (x \wedge \text{false}) \wedge (x \wedge \text{true}) = \text{false} \wedge x = \text{false}$ ⁵.

3 Intuitive Example

Suppose we are interested in knowing which unifications of the form $X = f(Y_1, \dots, Y_n)$ can be compiled into *deconstructions*, i.e. whose left hand side is ground⁶. We call these unifications *deconstructions*. Consider the code of `append` in normal form.

```
append(X, Y, Z) :-
  ( 1 X = [], 2 Z = Y
  ; 3 X = [H | Xt], 4 Z = [H | Zt], 5 append(Xt, Y, Zt) ).
```

Our goal is to describe all the call substitutions for `append` for which some or all of the unifications in `append` or in its recursive call are deconstructions. We assume the usual left-to-right resolution scheme.

For this example we need domains in which we can capture groundness information. For the concrete domain, we use $\wp(\text{ESubst})$ (Cf. Example 1). For the abstract domain, we can either use *Def* [12] – the set of definite boolean functions, or the more precise domain *Pos* [16] – the set of positive boolean expressions. Both domains, extended with a bottom element `false`, are complete lattices: $\langle \text{Def}, \sqsubseteq, \vee, \wedge, \text{true}, \text{false} \rangle$ and $\langle \text{Pos}, \sqsubseteq, \vee, \wedge, \text{true}, \text{false} \rangle$. The concretisation function γ introduced in Example 1 can be used for *Def* as well as *Pos*. Recall that `false` reflects the empty set of variable substitutions \emptyset which reflects a failing derivation, while $\gamma \text{true} = \wp(\text{ESubst})$.

Both domains can be used to represent goal-independent abstract groundness annotations. The goal-independent annotation table for the program consisting of `append` is in this case the same for *Def* as for *Pos*, and is depicted in Figure 2. This table is safe w.r.t. $\wp(\text{ESubst})$ and the combination operation \wedge . Indeed, if

⁵ Universal projection in *Pos* consists of replacing each of the variables that are projected out by the truth-values `false` and `true`, and then conjoining each of the resulting boolean expressions.

⁶ This is a pure hypothetical setting, and has no other purpose than to illustrate our approach.

$\delta \in Pos$ is a call substitution, then $\iota_i \wedge \delta$ is a safe description of the groundness information, for each individual program point i within `append`⁷.

pp	ι_{pp}
1	true
2	x
3	true
4	$x \leftrightarrow h \wedge x_t$
5	$(x \leftrightarrow h \wedge x_t) \wedge (z \leftrightarrow h \wedge z_t)$

Fig. 2. Goal-independent annotation table in *Pos* and *Def*.

pp	iter1		iter2	
	μ_{pp}	δ_{pp}^m	μ_{pp}	δ_{pp}^m
1	x	x	x	x
2	—	—	—	—
3	x	x	x	x
4	z	z	z	z
5	—	—	true	true
	$\vee\{x, z\} = \text{true}$		true	

Fig. 3. Gathered *Def*-information. μ_{pp} are minimal requirements, δ_{pp}^m are call requirements.

pp	iter1		iter2	
	μ_{pp}	δ_{pp}^m	μ_{pp}	δ_{pp}^m
1	x	x	x	x
2	—	—	—	—
3	x	x	x	x
4	z	z	z	z
5	—	—	$x_t \vee z_t$	$x \vee z$
	$\vee\{x, z\} = x \vee z$		$x \vee z$	

Fig. 4. Gathered *Pos*-information. μ_{pp} are minimal requirements, δ_{pp}^m are call requirements.

Before we can relate the optimisation of interest with the set of call substitutions for which it is safe, we need to formalise the condition under which a literal is subject for optimisation. In this example, $X = f(Y_1, \dots, Y_n)$ is a deconstruction if X is ground. In *Pos* or *Def*, this means that any call substitution δ for which $\delta \sqsubseteq x$ (because X needs to be ground) allows for the optimisation of the unification. The condition, X must be ground (represented by the abstract substitution x), is called the *minimal requirement* for optimising $X = f(Y_1, \dots, Y_n)$. We denote minimal requirements by the letter μ . The columns in Figure 3 and Figure 4 labelled μ_{pp} (under “iter 1”) show the (renamed) minimal requirements for the unifications to become deconstructions.

Given the minimal requirements expressed at the level of the individual unifications, we can now start reasoning about how these minimal requirements can be translated to call substitutions for `append`.

⁷ This is true for the specific case of a left-to-right selection rule. For other selection rules, other goal-independent annotations would have been obtained.

Consider a call substitution δ for `append`. From the safety of the goal-independent annotation table, we can check whether the unification at program point 3 is a deconstruction by verifying the formula

$$\iota_i \wedge \delta \sqsubseteq \mu_i \tag{1}$$

for $i = 3$. We can use (1) for deriving the *most general* call substitution δ'_i for which μ_i is satisfied and thus for which the optimisation is safe: every call substitution δ for which $\delta \sqsubseteq \delta'_i$ automatically satisfies (1). In the context of *Pos* computing this *most general* call substitution consists of computing the *pseudo-complement* of ι_i w.r.t. μ_i which corresponds to the logical implication [8]: $\delta'_i \stackrel{\text{def}}{=} \iota_i \rightarrow \mu_i$. For $i = 3$, we have $\iota_3 = \text{true}$, $\mu_3 = x$, and therefore $\delta'_3 = x$. This corresponds with our intuition that for X to be ground at program point 3, `append` must be called with a substitution in which at least X is ground. In general these pseudo-complements may involve any variable of the procedure (without giving the details for the (complex) formula for δ'_4 , we have $\text{Vars}(\delta'_4) = \{x, h, x_i, z\}$), while we are only interested in information regarding the variables occurring in the head of the predicate. Hence, a projection operation is needed. In this case we need universal projection, as we want a call substitution to be approximated by the computed pseudo-complement, for all possible values of the local variables (i.e. variables not appearing in the head of the procedure) in the procedure. We call these projected pseudo-complements *call requirements*, as they describe a requirement in terms of the variables of the head of the procedure such that if that requirement is fulfilled, optimisation of the literal within the procedure is safe. Figure 3 and Figure 4 list the call requirements for each of the unifications of interest (column “iter 1”, δ_{pp}^m). The call requirements can have two interesting extreme values: `true` — the optimisation is always possible as all concrete substitutions can be approximated by `true`, and `false` — there are no concrete substitutions that can be approximated by `false`, therefore, within the limits of the precision of the abstract domain, the optimisation is never safe. Note that in the abstract domain *Def*, not all abstract substitutions have a pseudo-complement, so in some cases approximation may be needed.

From Figure 3 and Figure 4 we can deduce that some of the unifications within `append(X, Y, Z)` become deconstructions if either X or Z is ground (column “iter 1”, δ_{pp}^m). We can now use this information to check when the recursive call in `append` might have deconstructions. In general, this process requires a fix-point computation, and therefore, to insure finiteness of the analysis, we take the minimal requirement for optimising a predicate call to be the least upper bound of the call requirements of its body literals. The idea is that a predicate call can be optimised if one of the literals in the body can be optimised. Another possibility is the use of the greatest lower bound for combining call requirements. Yet, in such a case we obtain a too restrictive minimal requirements as optimisation is only allowed if *all* the literals for which optimisation was detected can be optimised. A viable less restricting alternative is presented in Section 5.

From the individual call requirements x and z , we obtain the minimal requirement $x \vee z$ for a call to `append` in *Pos* – if X or Z is ground, optimisa-

tion may be possible – but true in *Def* – any call may have some optimisation possible⁸. Using this minimal requirement, we can again use (1) to compute a new call requirement: in *Pos*, at program point 5, the pseudo-complement of $\iota_5 = (x \leftrightarrow h \wedge x_t) \wedge (z \leftrightarrow h \wedge z_t)$ w.r.t. $\mu_5 = x_t \vee z_t$ (notice the renaming) is $\delta'_5 = x \vee z \vee x_t \vee z_t$, which, after projection, yields $\delta_5^m = \delta'_5|_{\{x,y,z\}} = x \vee z$.

The result of collecting the optimisations of `append` can be read from the column labelled δ_{pp}^m (“iter 2”) and is as follows: call substitutions in which X is ground allow the optimisation of the literals at program points 1, 3 and 5; call substitutions in which Z is ground allow the optimisation of the literals at program points 4 and 5. In *Def*, by overestimation, we also obtain that the recursive call of `append` (program point 5) might be optimisable for *any* call substitution, regardless of the groundness in that substitution.

The formalisation and generalisation of these ideas are presented in the following section.

4 Optimisation Derivation System

4.1 Basic Components

The optimisation derivation system starts with a clear specification of the intended optimisation, i.e. by pre-defining clear conditions about when some literals can be optimised. We call the literals for which such conditions are pre-defined the *base atoms*. The literals for which no such conditions are pre-defined are called *non base atoms*. If a non base atom is a builtin, then it is called a *non base builtin atom*. In the other case, i.e. the literal is a call to a user defined procedure, it is called a *non base call atom*. In essence, we use the following subdivision for the set of literals:

$$Literal = BaseAtom \cup NBBuiltin \cup NBCall$$

where *BaseAtom*, *NBBuiltin* and *NBCall* denote the set of base atoms, the set of non base builtin atoms and the set of non base call atoms respectively. This subdivision is important because optimisation information is *generated* at base atoms and *propagated* through non base call atoms. Non base builtin atoms are neutral w.r.t. to the intended optimisations. For simplicity, we use the notation $p(X_1, \dots, X_n)$, where $\{X_1, \dots, X_n\} \subseteq \mathcal{V}$ and $p \in \Pi$, to refer to literals, therefore not distinguishing unifications from other literals.

For each program we assume that a goal-independent annotation table is available. This annotation table is expressed in terms of an abstract Noetherian domain $\langle \mathcal{A}, \sqsubseteq, \sqcup, \sqcap, \top_{\mathcal{A}}, \perp_{\mathcal{A}} \rangle$. The annotation table is considered safe w.r.t. a concrete domain $\langle \mathcal{C}, \subseteq, \cup, \cap, \top_{\mathcal{C}}, \perp_{\mathcal{C}} \rangle$ and an abstract combination operator $\otimes : \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{A}$.

The abstract domain \mathcal{A} in terms of which the goal-independent information is expressed must be suitable to express the so-called *minimal requirements* for optimisation for literals in the language.

⁸ While $x \vee z$ is a fair estimation, true is a clear overestimation, which illustrates how the precision of the domain can influence the obtained results.

Definition 3 (Requirement, Minimal Requirement). A requirement for optimisation of a literal $p(X_1, \dots, X_n)$ is a substitution $\mu' \in \mathcal{A}$, such that for all concrete substitutions described by μ' , optimisation of the literal is allowed. Formally: $\forall \sigma \in \mathcal{C}$, if $\sigma \sqsubseteq \gamma \mu'$ then $p(X_1, \dots, X_n)$ can be optimised. The minimal requirement for optimisation of a literal $p(X_1, \dots, X_n)$, denoted by μ , is the largest requirement w.r.t. to the ordering \sqsubseteq of the abstract domain. It is called minimal in the sense that it describes the largest set of concrete substitutions for which optimisation is possible, hence imposes the least restrictions on these substitutions.

Note that if μ is a minimal requirement for a literal, then the literal can be optimised $\forall \delta \in \mathcal{A}$ for which $\delta \sqsubseteq \mu$. This is a consequence of the monotonicity of γ . In the following sections, we mainly use this formulation.

The minimal requirements for base atoms are predefined and recorded in a so-called *base table*.

Definition 4 (Base Table). A base table, denoted by the letter B , is a function mapping base atoms (modulo variance) on their minimal requirements $BaseTable : BaseAtom_{/\approx} \rightarrow \mathcal{A}$.

The goal of the optimisation derivation system is to derive minimal requirements for non base call atoms. To make the distinction between the minimal requirement of a non base call atom which expresses when *some* optimisation in the called predicate can be allowed, with the minimal requirement which expresses when some specific literal in the called predicate can be optimised, we introduce the notion of *call requirement*:

Definition 5 (Call Requirement). A call requirement for a predicate p with respect to a program point i is an abstract substitution denoted by δ^m , such that for each concrete call to p described by δ^m , the optimisation of the literal at program point i in the definition of p is allowed. Formally: $\forall \sigma \in \mathcal{C}$, if $\sigma \sqsubseteq \gamma \delta^m$, then l_i in p can be optimised. A call requirement is minimal if there does not exist a larger call requirement w.r.t. the order relation \sqsubseteq in \mathcal{A} . In the following we use call requirement as a synonym for minimal call requirement.

Finally, the abstract domain is required to have a *generalised pseudo-complement* operator which is defined with respect to the combination operator used for the goal-independent annotations.

Definition 6 (Generalised Pseudo-Complement). The generalised pseudo-complement, denoted by the function $\overset{\otimes}{\rightarrow} : \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{A}$, of an abstract substitution δ_a relative to an abstract substitution δ_b and w.r.t. the combination operator $\otimes : \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{A}$ is defined as:

$$\begin{aligned} \text{let } \delta'_c &= \bigsqcup \{ \delta_c \in \mathcal{A} \mid \delta_a \otimes \delta_c \sqsubseteq \delta_b \} \\ \text{then } \delta_a \overset{\otimes}{\rightarrow} \delta_b &= \delta'_c \text{ iff } \delta_a \otimes \delta'_c \sqsubseteq \delta_b \end{aligned}$$

This is a generalisation of the definition of a *pseudo-complement* in the sense that it is defined in terms of a general combination operator \otimes instead of the greatest lower bound operation \sqcap of the abstract domain [8].

If the concrete domain contains a bottom element $\perp_{\mathcal{C}}$ which represents failure, i.e. a failing derivation, then by the strictness of the concretisation function, we know that if the generalised pseudo-complement is $\perp_{\mathcal{A}}$, then the optimisation is *never* possible as there is no non-failing concrete substitution which is correctly described by $\perp_{\mathcal{A}}$. Therefore, by requiring that $\perp_{\mathcal{C}}$ represents a non-existing concrete environment (failure)⁹, we can guarantee that $\{c \in \mathcal{A} \mid a \otimes c \sqsubseteq b\}$ in Definition 6 is never empty as it will always at least contain $\perp_{\mathcal{A}}$. This makes the definition of the pseudo-complement and its use more compact as there is no further discussion needed on whether this set is empty or not, i.e. on whether there exists at least one call substitution δ_c for which $\delta_a \otimes \delta_c \sqsubseteq \delta_b$ is satisfied or not. Whether or not δ'_c satisfies $\delta_a \otimes \delta'_c \sqsubseteq \delta_b$ depends on the abstract domain, and in some cases approximation may be needed. This issue is discussed in Section 6.

4.2 Framework

In the process of deriving optimisations we compute call requirements from minimal requirements. We do this by using the generalised pseudo-complements of the goal-independent annotations of a literal with respect to the minimal requirement defined or computed for that literal.

Using the above basic components, we can now formalise the actual optimisation derivation process. We do this using denotational semantics [22, 1], which relates optimisation opportunities to the syntactic objects constituting a program.

We represent the optimisation opportunities of a program as a table which, per predicate, lists the optimisation opportunities that are possible within the procedure defining the predicate. Each optimisation opportunity is characterised by a program point $i \in \text{pp}$, referring to the literal that can be optimised, and a call requirement $\delta^m \in \mathcal{A}$ on the defined procedure for which that literal can be optimised. Therefore, the optimisation possibilities of an individual procedure are expressed as a *local optimisation table*, which is a function of type $LocalOptTable : \text{pp} \rightarrow \mathcal{A}$. The optimisation possibilities of a program are recorded in a so-called *optimisation table*, which can be seen as a function $OptTable : Procedure \rightarrow LocalOptTable$ mapping a procedure on a local optimisation table characterising the optimisation possibilities of that procedure.

The intended meaning of the optimisation table is defined as follows.

Definition 7 (Intended meaning of $OptTable$). *Let $\Omega : OptTable$ be the result of interpreting the program consisting of the procedures p_1 to p_{n_p} . Let $(p_j, \omega_j) \in \Omega$ and $(i, \delta_i^m) \in \omega_j$. If a call substitution δ of p_j is such that $\delta \sqsubseteq \delta_i^m$, then, depending on the nature of the literal at program point i , i.e. l_i , we interpret this as:*

⁹ This is not a restriction. If the domain didn't contain $\perp_{\mathcal{C}}$ as failure, then it is always possible to extend the domain with such an element such that $\perp_{\mathcal{C}} \sqsubseteq \sigma, \forall \sigma \in \mathcal{C}$.

- if $l_i \in \text{BaseAtom}$, then the literal can definitely be optimised for calls with call substitution δ .
- otherwise, i.e. $l_i \in \text{NBCall}$, the procedure corresponding with the called atom might allow some optimisations within it.

This means that we expect *strong* results for base atoms, while the results for non base call atoms are allowed to be *weak*.

Figure 6 defines the semantic functions for each individual syntactic object that is part of our language. The signatures of these functions, as well as the types of the entities used in these functions are defined in Figure 5. In Figure 6 we use the notation $\Omega(p(\overline{Y}))$ to refer to the local optimisation table recorded for the procedure $p(\overline{Y}) \leftarrow g$.

$$\begin{aligned}
\text{LocalOptTable} &= \text{pp} \rightarrow \mathcal{A} \\
\text{OptTable} &= \text{Literal} \rightarrow \text{LocalOptTable} \\
\text{Ann} &= \text{pp} \rightarrow \mathcal{A} \\
\text{BaseTable} &= \text{BaseAtom} \rightarrow \mathcal{A} \\
\mathbf{P} &: \text{Program} \rightarrow \text{BaseTable} \rightarrow \text{Ann} \rightarrow \text{OptTable} \\
\mathbf{F} &: \text{Program} \rightarrow \text{BaseTable} \rightarrow \text{Ann} \rightarrow \text{OptTable} \rightarrow \text{OptTable} \\
\mathbf{Pr} &: \text{Procedure} \rightarrow \text{BaseTable} \rightarrow \text{Ann} \rightarrow \text{OptTable} \rightarrow \text{OptTable} \\
\mathbf{G} &: \text{Goal} \rightarrow \text{BaseTable} \rightarrow \text{Ann} \rightarrow \wp(\mathcal{V}) \rightarrow \\
&\quad \text{OptTable} \rightarrow \text{LocalOptTable} \rightarrow \text{LocalOptTable} \\
\mathbf{L} &: \text{Literal} \rightarrow \text{BaseTable} \rightarrow \text{Ann} \rightarrow \wp(\mathcal{V}) \rightarrow \\
&\quad \text{OptTable} \rightarrow \text{LocalOptTable} \rightarrow \text{LocalOptTable}
\end{aligned}$$

Fig. 5. Types and signatures used in the semantic functions.

The optimisation table of a program is defined w.r.t. a base table B and a goal-independent annotation table A . Both tables are predefined. For the base table this could be done manually, while in most cases the annotation table will have been derived by a preceding analysis. The semantics is defined as the least fixed point of $\mathbf{F}[[r]]BA$ which maps an initial optimisation table on a new optimisation table. It is defined in terms of the individual optimisation possibilities of each of the procedures in r . The optimisations possible within a procedure are given in terms of the optimisations within the goal defining the procedure. Note that the optimisations of the goal are defined w.r.t. an empty local optimisation table $\perp_\omega = \{(i, \perp_{\mathcal{A}}) \mid i \in \text{pp}(p)\}$, where p is the procedure to which the local optimisation table belongs. In essence, the semantics of a goal is defined as the union of all the local optimisation criteria coming from the literals that define that goal.

The optimisation opportunities for a literal are different depending on whether it is a base atom, a non base call atom, or a non base builtin atom. For base atoms, we compute the generalised pseudo-complement of the goal-independent annotation relative to the (renamed) minimal requirement for that base atom. This pseudo-complement is projected on the variables used in the head atom of

$$\begin{aligned}
\mathbf{P}[r]BA &= \text{lfp}(\mathbf{F}[r]BA) \\
\mathbf{F}[p_1, \dots, p_i(\overline{X}) \leftarrow g, \dots, p_{n_p}]BA\Omega p_i(\overline{Y}) &= \mathbf{Pr}[p_i(\overline{X}) \leftarrow g]BA\Omega p_i(\overline{Y}) \\
\mathbf{Pr}[p(\overline{X}) \leftarrow g]BA\Omega p(\overline{Y}) &= \rho_{\overline{X} \rightarrow \overline{Y}}(\mathbf{G}[g]BA\{\overline{X}\}\Omega \perp_\omega) \\
\mathbf{G}[g_1, g_2]BA\mathcal{H}\Omega\omega &= \text{let } \omega_1 = \mathbf{G}[g_1]BA\mathcal{H}\Omega\omega \text{ in} \\
&\quad \text{let } \omega_2 = \mathbf{G}[g_2]BA\mathcal{H}\Omega\omega \text{ in} \\
&\quad \omega_1 \cup \omega_2 \\
\mathbf{G}[g_1; g_2]BA\mathcal{H}\Omega\omega &= \text{let } \omega_1 = \mathbf{G}[g_1]BA\mathcal{H}\Omega\omega \text{ in} \\
&\quad \text{let } \omega_2 = \mathbf{G}[g_2]BA\mathcal{H}\Omega\omega \text{ in} \\
&\quad \omega_1 \cup \omega_2 \\
\mathbf{G}[\text{if } g_1 \text{ then } g_2 \text{ else } g_3]BA\mathcal{H}\Omega\omega &= \text{let } \omega_i = \mathbf{G}[g_i]BA\mathcal{H}\Omega\omega, \ i = 1, 2, 3 \text{ in} \\
&\quad \omega_1 \cup \omega_2 \cup \omega_3 \\
\mathbf{G}[\text{not } g]BA\mathcal{H}\Omega\omega &= \mathbf{G}[g]BA\mathcal{H}\Omega\omega \\
\mathbf{G}[l]BA\mathcal{H}\Omega\omega &= \mathbf{L}[l]BA\mathcal{H}\Omega\omega \\
\mathbf{L}[p(\overline{X})]BA\mathcal{H}\Omega\omega &= \text{let } \iota = A(\text{pp}(p(\overline{X}))) \text{ in} \\
&\quad \text{let } (p(\overline{Y}), \mu_{p_Y}) \in B \text{ in} \\
&\quad \text{let } \mu_p = \rho_{\overline{Y} \rightarrow \overline{X}}(\mu_{p_Y}) \text{ in} \\
&\quad \text{let } \delta^m = (\iota \xrightarrow{\otimes} \mu_p)|_{\mathcal{H}} \text{ in} \\
&\quad \omega \cup \{(\text{pp}(p(\overline{X}))), \delta^m\} \\
&\quad \text{where } p(\overline{X}) \in \text{BaseAtom} \\
\mathbf{L}[p(\overline{X})]BA\mathcal{H}\Omega\omega &= \text{let } \iota = A(\text{pp}(p(\overline{X}))) \text{ in} \\
&\quad \text{let } \mu_{p_Y} = \sqcup\{\mu'_{p_Y} \mid (i, \mu'_{p_Y}) \in \Omega(p(\overline{Y}))\} \text{ in} \\
&\quad \text{let } \mu_p = \rho_{\overline{Y} \rightarrow \overline{X}}(\mu_{p_Y}) \text{ in} \\
&\quad \text{let } \delta^m = (\iota \xrightarrow{\otimes} \mu_p)|_{\mathcal{H}} \text{ in} \\
&\quad \omega \cup \{(\text{pp}(p(\overline{X}))), \delta^m\} \\
&\quad \text{where } p(\overline{X}) \in \text{NBCall} \\
\mathbf{L}[p(\overline{X})]BA\mathcal{H}\Omega\omega &= \omega \\
&\quad \text{where } p(\overline{X}) \in \text{NBBuiltin}
\end{aligned}$$

Fig. 6. Semantic Functions for the Optimisation Environments.

the procedure to which the literal belongs, and the result, the call requirement δ^m , is combined with the program point of the literal, and added to the local optimisation table that has been built up to this literal.

If the literal is a non base call atom, then the local optimisation table for the procedure defining the called atom is consulted. All the call requirements within this table are combined (using the least upper bound) into one single minimal requirement for optimisation. The result μ_p is used in the same way as the minimal requirement is used for base atoms.

Finally, non base builtin atoms introduce no new optimisations.

For this semantics to be acceptable we need to prove that it is well-defined, i.e. can be computed by a terminating fixpoint computation [22], and correct with respect to the intended meaning of the derived optimisation table we gave earlier. Before we can prove this, we prove the following lemma:

Lemma 1. *The generalised pseudo-complement \boxtimes is monotone in its second argument.*

Proof. Let $c_1 = a \boxtimes b_1$, $c_2 = a \boxtimes b_2$, and $b_1 \sqsubseteq b_2$. We need to prove that $c_1 \sqsubseteq c_2$. By the definition of \boxtimes we have $a \otimes c_1 \sqsubseteq b_1$, and therefore by transitivity $a \otimes c_1 \sqsubseteq b_2$. This means that $c_1 \in C_2 = \{c \mid a \otimes c \sqsubseteq b_2\}$. By definition we have $c_2 = \bigsqcup C_2$, and therefore $c_1 \sqsubseteq c_2$. \square

We can now prove the central theorem of this paper:

Theorem 1. *The semantic function \mathbf{F} is well-defined and correct w.r.t. Definition 7.*

Proof. Given our limitation to Noetherian domains it suffices to show that $\mathbf{F}[r]BA$ is monotone in its *OptTable*-argument in order to conclude that the semantics is well-defined (terminating).

If $p \in \text{Procedure}$, we define $\perp_{\omega,p} = \{(i, \perp_{\mathcal{A}}) \mid i \in \text{pp}(p), l_i \in \text{BaseTable} \cup \text{NBCall}\}$. If $\omega_1, \omega_2 \in \text{LocalOptTable}$, then $\omega_1 \sqsubseteq \omega_2$ iff $\forall (i, \delta_{i,1}^m) \in \omega_1, \exists (i, \delta_{i,2}^m) \in \omega_2$ such that $\delta_{i,1}^m \sqsubseteq \delta_{i,2}^m$. Similarly, $\perp_{\Omega} = \{(p, \perp_{\omega,p}) \mid p \in \text{Procedure}\}$ and if $\Omega_1, \Omega_2 \in \text{OptTable}$, then $\Omega_1 \sqsubseteq \Omega_2$ iff $\forall (p, \omega_{p,1}) \in \Omega_1, \exists (p, \omega_{p,2}) \in \Omega_2$ such that $\omega_{p,1} \sqsubseteq \omega_{p,2}$.

As the minimal requirements are *generated* at the level of literals, it suffices to show that if $\Omega_1 \sqsubseteq \Omega_2$, then $\mathbf{L}[p(\bar{X})]BA\mathcal{H}\Omega_1\omega \sqsubseteq \mathbf{L}[p(\bar{X})]BA\mathcal{H}\Omega_2\omega$. The only non-trivial case is where $p(\bar{X}) \in \text{NBCall}$.

We subscribe the intermediate values in the definition of \mathbf{L} with 1 or 2, depending on whether we refer to \mathbf{L} with respect to Ω_1 , or Ω_2 . In the definition of $\mathbf{L}[p(\bar{X})]$ for $p(\bar{X}) \in \text{NBCall}$, we have $\mu_{p_Y,1} \sqsubseteq \mu_{p_Y,2}$ (monotonicity of \sqcup), thus $\mu_{p,1} \sqsubseteq \mu_{p,2}$. According to Lemma 1 and the monotonicity of the projection operation, we have $\delta_1^m \sqsubseteq \delta_2^m$, which proves the monotonicity of \mathbf{L} w.r.t. Ω .

Finally, the correctness of the semantics with respect to the definition of the intended meaning of a logic program follows from the correctness of the base table, the safeness of the goal-independent annotation table, and the definition of \mathbf{L} for base atoms and non base call atoms. The latter depends on the definition of the generalised pseudo-complement. This definition guarantees that if the requirement μ is minimal, then the call requirement δ^m computed from it will be minimal too. While for base atoms we use the exact minimal requirements as tabled in the base table, the minimal requirements used for non base call atoms might be approximations due to the use of the \sqcup in the definition of \mathbf{L} . \square

Given the fact that each monotone function over a Noetherian lattice is automatically continuous [17], we can use the Kleene sequence [15] to compute the fixpoint of the semantics: $\mathbf{P}[r]BA = (\mathbf{F}[r]BA)^n \perp_{\Omega}$.

5 Improved Optimisation Derivation System

In the above formalisation, the precision of the optimisation information is lost at non base call atoms where we take the least upper bound of all the call

requirements, and compute one single generalised pseudo-complement of the resulting minimal requirement. One possibility for avoiding this loss of precision is by collecting the individual call requirements as a set. The semantics of a procedure would then be given by a function $LocalOptTable^\wp = \text{pp} \rightarrow \wp(\mathcal{A})$ instead of $LocalOptTable = \text{pp} \rightarrow \mathcal{A}$. In such a semantics, where each semantic function \mathbf{A} in Figure 6 is replaced by a corresponding semantic function \mathbf{A}^\wp , and for which each occurrence of the type $LocalOptTable$ is replaced by the type $LocalOptTable^\wp$, most of the definitions remain the same as for the derivation of simple optimisations, except the definition of the semantics of base atoms and non base call atoms. These are defined in Figure 7.

$$\begin{aligned}
\mathbf{L}^\wp[p(\overline{X})]BA\mathcal{H}\Omega^\wp\omega^\wp &= \text{let } \iota = A(\text{pp}(p(\overline{X}))) \text{ in} \\
&\quad \text{let } (p(\overline{Y}), \mu_{p_Y}) \in B \text{ in} \\
&\quad \text{let } \mu_p = \rho_{\overline{Y} \rightarrow \overline{X}}(\mu_{p_Y}) \text{ in} \\
&\quad \text{let } \delta^m = (\iota \overset{\otimes}{\rightarrow} \mu_p)|_{\mathcal{H}} \text{ in} \\
&\quad \omega^\wp \cup \{(\text{pp}(p(\overline{X}))), \{\delta^m\}\} \\
&\hspace{15em} \text{where } p(\overline{X}) \in BaseAtom \\
\mathbf{L}^\wp[p(\overline{X})]BA\mathcal{H}\Omega^\wp\omega^\wp &= \text{let } \iota = A(\text{pp}(p(\overline{X}))) \text{ in} \\
&\quad \text{let } \omega_{p_Y}^\wp = \Omega^\wp(p(\overline{Y})) \text{ in} \\
&\quad \text{let } M_{p_Y} = \bigcup \{M_{p_Y}^* \mid (i, M_{p_Y}^*) \in \omega_{p_Y}^\wp\} \text{ in} \\
&\quad \text{let } M_p = \{\mu_p \mid \mu_{p_Y} \in M_{p_Y} \wedge \mu_p = \rho_{\overline{Y} \rightarrow \overline{X}}(\mu_{p_Y})\} \text{ in} \\
&\quad \text{let } \Delta^m = \{\delta^m \mid \mu_p \in M_p \wedge \delta^m = (\iota \overset{\otimes}{\rightarrow} \mu_p)|_{\mathcal{H}}\} \text{ in} \\
&\quad \omega^\wp \cup (\text{pp}(p(\overline{X})), \Delta^m) \\
&\hspace{15em} \text{where } p(\overline{X}) \in NB\mathcal{C}all
\end{aligned}$$

Fig. 7. Definition of the semantics of literals when collecting sets of optimisations.

The optimisation opportunities for base atoms remain almost the same, except that we store the call requirement δ^m as the singleton set $\{\delta^m\}$. In the definition of the semantics of a non base call atom, we consult the intermediate optimisation table Ω^\wp , extract all the call requirements that are defined for the called procedure, and take the union of all these call requirements (instead of the least upper bound as in the simple semantics). Each call requirement in this set is then renamed and pseudo-complemented. The result, Δ^m , is recorded in the local optimisation table.

The intended meaning of the resulting optimisation table is similar to the meaning we gave in Definition 7, except that now, the optimisation information derived for non base call atoms is also *definite*. If a call substitution δ of a procedure p is such that there exists a call requirement within p , δ_i^m ($i \in \text{pp}$), for which $\delta \sqsubseteq \delta_i^m$, then the literal at that program point i can *definitely* be optimised, regardless of the nature of that literal. Yet, a verification step is still required in order to check what exactly can be optimised if the literal is a non base call atom.

It is straightforward to show that, *if a fixpoint is reached*, the semantics \mathbf{P}^φ is correct with respect to this intended meaning. But in this setting, we cannot guarantee that for each Noetherian domain a fixpoint will always be reached. Either extra restrictions on the domain, such as finiteness, need to be imposed, or widening operations need to be used [6, 7, 26, 3].

6 Discussion

In the above semantics we have restricted the language to non-modular programs. In general, we can easily lift this restriction by computing all the optimisation tables for the modules within a program in a bottom-up way: if a module A depends on a module B , then the optimisation derivation system should first derive the optimisations for the procedures in B , before deriving the optimisations possible within module A . In such cases, we have no loss of precision. Note that the optimisation derivation system is not concerned with the actual version generation. This still requires a separate pass, which becomes indeed more complicated in the presence of modules. Programs with circular dependencies among modules pose the usual problems as ideally the fixpoint iteration should be done over the module graph [23].

We introduced *minimal requirements*. These are call substitutions μ related to literals l , such that for each call substitution δ , if $\delta \sqsubseteq \mu$, then l can be optimised. This means that we need an abstract domain in which a suitable order relation \sqsubseteq can be expressed for the intended optimisation. A careful design of the abstract domain is therefore necessary.

We generalised the notion of pseudo-complement such that the operation is not limited to the greatest lower bound of the abstract domain used. We expect that for most domains, the combination operation will indeed be the greatest lower bound, but we did not want to limit our formalisation to this operation. One reason is that we plan to use the optimisation derivation system for our CTGC system. Given the fact that the abstract domain \mathcal{A} used in the context of CTGC is a complex domain composed of alias information and in use information, the combination operation is not simply the greatest lower bound, but another, more complex operation. In this context, we also want to be able to reason about the *largest* abstract substitution δ for which the result of combining it with a goal-independent annotation can still correctly be approximated by the minimal requirement for optimisation. Computing this *largest* abstract substitution is exactly what the pseudo-complement does with respect to \sqcap . Therefore, it is natural to generalise this operation to allow other combination operations.

In the present setting we assumed that the abstract domain is such that the (generalised) pseudo-complement of two elements always exists. This means that in Definition 6, δ'_c always satisfies the formula $\delta_a \otimes \delta'_c \sqsubseteq \delta_b$. This imposes a certain restriction on the domains that can be used. In our intuitive example, we suggested that approximations of these pseudo-complements can be a good alternative for situations and domains for which the exact pseudo-complement

does not exist (which is for example the case for substitutions in *Def*). For these approximations, there are two alternatives. Either one approximates the exact pseudo-complement, say δ^m , *from above* — $\delta^m \sqsubseteq \overline{\delta^m}$, or *from below* — $\underline{\delta^m} \sqsubseteq \delta^m$. Approximating from above means that there might be some call substitutions, which, when compared to $\overline{\delta^m}$, erroneously suggest that optimisation is possible. This means that even for base atoms, we cannot be sure how to interpret the obtained optimisation results. Therefore, this alternative is not interesting. On the other hand, approximating from below respects safety of the results for base atoms, yet, in some cases, some optimisations might not be spotted. Note that such approximation might impose other restrictions on the abstract domain used. Determining exactly which restrictions requires further investigation.

Finally, note that in our optimisation system we have deliberately taken the least upper bound of the individual call requirements within a procedure instead of the greatest lower bound. The advantage of this approach is that we can spot all possible optimisations. The disadvantage is that if the optimisation derivation system spots that a procedure call might be optimisable, we do not know exactly which optimisations are possible, if at all (this may be the case for some abstract domains). The price to pay is that extra checks are necessary during the actual version generation pass. The alternative, the greatest lower bound, has the advantage that if the optimisation derivation spots optimisations, then these optimisations are definitely possible. The disadvantage though is that an optimisation is only identified as such if *all* the optimisations within the called procedure are safe. Hence, a lot of intermediate combinations of optimisations are simply missed, which is exactly what we want to avoid. A possible solution to the loss of precision using \sqcup instead of \sqcap is by collecting the call requirements as sets, keeping them distinct, as we described in Section 5.

7 Related Work

Program analyses which are part of optimising compilers are always confronted with the issue of version generation: if a predicate can be optimised in several ways, which versions should be generated knowing that not all optimisations are safe in each calling context? Without a priori information about the use of the predicates, there are two possibilities: either the optimising compiler generates all possible combinations of optimised predicates, or the compiler uses some predefined criterion which guides the number of versions that it will produce (a common heuristical criterion being that at most two versions are generated per predicate: a genuine, non-optimised version, and a fully optimised version). While the former ensures that the resulting program can make full use of its optimisation potential, the code explosion it requires may make this technique unfeasible in practice. The alternative option limits the code explosion, but it also limits the optimisation possibilities. It is possible to overcome the above problems of code explosion and suboptimality by using information about how some of the predicates are used. Up to now, the most common technique was to build a top-down analysis and version generation process. In such a system,

each call to a predicate is described in the abstract domain, and for each such call a different optimised version is generated. Depending on the granularity of the abstract domain, this may still involve the creation of a large number of versions [25, 14, 11, 24]. In [24], for example, the analysis and version generation pass is therefore followed by a pruning pass: this pass tries to identify similar versions, and merges them, hence reducing the number of versions (but possibly also at the cost of producing a slightly suboptimal program). The pruning is based on heuristics. All the above mentioned techniques have one aspect in common: each of the versions is identified by one specific call substitution for which the procedure call is considered safe. If a different call substitution is encountered, then none of the analyses can safely decide whether it is still safe to call the optimised version or not.

In this paper we propose an optimisation derivation system which relies on information which is a priori collected (in a call independent way), and which identifies each single optimisation by a description of the call substitutions for which that optimisation is safe. This gives the version generation pass almost full knowledge about when the optimisations are possible, enabling more sophisticated heuristics for version generation. Also, given a new call to a procedure, it enables the version generation system to simply compare the call substitution with the description of call substitutions for which some optimisation is safe, hence safely identifying the possible optimisations and therefore the version to be used.

To derive the possible optimisations, we have introduced a new optimisation derivation system which is based on generalised pseudo-complements (for mapping individual minimal requirements to the variables appearing in the head atom of the procedure they belong to), and least upper bounds (for collecting call requirements) with the purpose of deriving a single description of when an optimisation is safe. These ideas are very similar to the ideas behind the so-called *backwards analysis* (as opposed to the classical *forward analyses*), a new analysis system introduced by King et al. [13]. The purpose of backwards analysis is also to derive a description of the call substitutions for which some predicate call is guaranteed not to fail (w.r.t. some criterion such as termination, moding, etc.). This could be transcribed to our context as describing the call substitutions for which some optimised predicate is guaranteed not to fail. Yet, there are two reasons why this model of analysis does not fully fit our needs. A first reason is that, per predicate, backwards analysis computes information on a program point i using information it computed a step earlier for program point $i + 1$, the program point associated with the *next* literal within the same procedure w.r.t. i , under the usual left-to-right resolution scheme. This is necessary if all the underlying basic information still needs to be derived, yet in our setting we consider this information already present. Therefore we want to translate the optimisation information at a program point to the head of the predicate definition, without visiting all the intermediate program points. Note that the operator which makes this backwards propagation of the information possible is the pseudo-complement. Our system is using a similar operation, the *generalised*

pseudo-complement. The second reason why our goal does not immediately fit within backwards analysis is that if a predicate contains more than one literal that can be optimised, then the call substitutions for each of these optimisations to occur are conjoined during backwards analysis. This results in a restrictive call substitution for which *all* optimisations are safe, instead of a substitution for which some *individual* optimisation may be done. Our goal was to derive the non-restrictive call substitution for optimisation.

8 Conclusion

Collecting information about possible optimisations within a program *before* actually generating code supporting the optimisations is, in our view, an essential step for a better understanding of which versions need to be generated to obtain better global optimisation results. Here we have developed a mechanism which makes it possible to relate the optimisations with the call substitutions for which they are safe. We intend to use it for our CTGC system. Implicitly, the current CTGC implementation [18] already uses a generalised pseudo-complement operator to translate local optimisation conditions to the variables in the head of a predicate. Given the fact that CTGC depends on a compound abstract domain (containing possible aliasing and in use information), even a brief and intuitive development of how to adapt CTGC to the proposed optimisation derivation system is beyond the scope of this paper.

References

1. Lloyd Allison. *A Practical Introduction to Denotational Semantics*. Cambridge Computer Science Texts. Cambridge-University-Press, 1986.
2. Roberto Bagnara, Enea Zaffanella, and Patricia M. Hill. Enhanced sharing analysis techniques: A comprehensive evaluation. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 103–114, Montreal, Canada, 2000. ACM Press.
3. Michael Codish, Andy Heaton, and Andy King. Widening pos for efficient and scalable groundness analysis of logic programs. Technical Report 16-97, University of Kent at Canterbury, December 1997.
4. Michael Codish, Harald Søndergaard, and Peter Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, 1977.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2 & 3):103–179, May/July 1992.
7. Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe

- and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, Leuven, Belgium, 1992. LNCS 631, Springer-Verlag.
8. Roberto Giacobazzi and Francesca Scozzari. A logical model for relational abstract domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.
 9. Andy Heaton, Muhamed Abo-Zaed, Michael Codish, and Andy King. Simple, efficient and scalable groundness analysis of logic programs. *The Journal of Logic Programming*, 45(1-3):143–156, 2000.
 10. Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Dept. of Computer Science, University of Melbourne, February 1996.
 11. Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Programming Languages and Systems — ESOP’94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 287–301. Springer-Verlag, 1994.
 12. Jacob M. Howe and Andy King. Implementing Groundness Analysis with Definite Boolean Functions. In G. Smolka, editor, *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, March 2000.
 13. Andy King and Lunjin Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2(4-5):517–547, July 2002.
 14. Michael Leuschel, Bern Martens, and Danny De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208 – 258, 1998.
 15. John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
 16. Kim Marriott and Harald Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Lett. Prog. Lang. Syst.*, 2(1-4):181–196, 1993.
 17. Kim Marriott, Harald Søndergaard, and Neil D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, May 1994.
 18. Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe. A module based analysis for memory reuse in Mercury. In J. Lloyd et al, editor, *Computational Logic - CL 2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1255–1269. Springer-Verlag, 2000.
 19. Nancy Mazur, Peter Ross, Gerda Janssens, and Maurice Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury. In Philippe Codognet, editor, *Proceedings of ICLP 2001 - Seventeenth International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
 20. Anne Mulkers. *Live Data Structures in Logic Programs, Derivation by Means of Abstract Interpretation*. Lecture Notes in Computer Science 675. Springer-Verlag, 1993.
 21. Anne Mulkers, Will Winsborough, and Maurice Bruynooghe. Analysis of shared data structures for compile-time garbage collection in logic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, 1990. MIT Press, Cambridge.
 22. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing. John Wiley & Sons, Inc., Chichester, 1992.

23. Germán Puebla and M. Hermenegildo. Some issues in analysis and specialization of modular Ciao-Prolog programs. In M. Leuschel, editor, *Proceedings of the Workshop on Optimization and Implementation of Declarative Languages*, Las Cruces, 1999. In *Electronic Notes in Theoretical Computer Science*, Volume 30 Issue No.2, Elsevier Science.
24. Germán Puebla and Manuel Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *Journal of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.
25. Wim Vanhoof and Maurice Bruynooghe. Binding-time analysis for Mercury. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*, pages 500–514. MIT Press, 1999.
26. Enea Zaffanella, Roberto Bagnara, and Patricia M. Hill. Widening Sharing. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 414–431, Paris, France, 1999. Springer-Verlag, Berlin.