Contextual Specialization of Constraint Logic Programs

Fabio Fioravanti, Maurizio Proietti

IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
{fioravanti,proietti}@iasi.rm.cnr.it

Abstract

We address the problem of specializing a constraint logic program w.r.t. a specific context of use. We consider constraints over a generic domain and we assume that the context of use of a program is specified by a constrained atom. We follow an approach based on transformation rules and strategies. We present a novel transformation rule, called contextual constraint replacement, to be combined with the traditional unfolding and folding rules. We show the correctness of the transformation rules w.r.t. least models and we present a general strategy for automating the application of these rules. We show through examples the power of our method and its superiority w.r.t. existing methods for partial deduction of constraint logic programs based on extensions of Lloyd and Shepherdson's approach.

1 Introduction

The goal of *program specialization* is the automatic adaptation of a program to the context where it is used. Several program transformation techniques have been proposed in the literature for achieving program specialization. One of the most well-known among these techniques is *partial evaluation* [10], which given a program and part of the input data, the so-called *static input*, derives a new, specialized program, which may be more efficient than the original one because it exploits the partial knowledge of the input.

Partial evaluation works by precomputing some results which depend on the static input only, so that the specialized program only has to perform computations that depend on the input data which are not known at partial evaluation time, that is, the *dynamic input*. In order to be meaningful, partial evaluation should be *correct*, that is, the specialized program applied to the dynamic input values should return the same output values as the original program applied to the complete (that is, static + dynamic) input values.

In this paper we consider the case of *constraint logic programs* [8] (CLP, for brevity) over a domain $\mathcal{D}$, and we formulate the problem of *specializing a program w.r.t. a constrained atomic query*, as follows: given a CLP program P, an atomic query $p(X)$, and a constraint c, we want to derive a new program P_s and a new query $p_s(X)$ such that for every element d in the domain $\mathcal{D}$ satisfying the constraint c, we have

$$p(d) \in lm(P, \mathcal{D}) \qquad \text{iff} \qquad p_s(d) \in lm(P_s, \mathcal{D})$$

where, for any program P and domain $\mathcal{D}$, $lm(P, \mathcal{D})$ denotes the *least $\mathcal{D}$-model* of P [9].

Thus, our program specialization problem is more general than the one usually considered by partial evaluation techniques, because we assume that the static input is specified by a constraint c which may denote a *possibly infinite set of values*, instead of a *single value*.

For instance, consider a CLP($\mathcal{R}$) program P containing the following clauses for a predicate p:

$p(X) \leftarrow X \geq 0,\ q(X)$
$p(X) \leftarrow X \leq 0,\ r(X)$
$p(X) \leftarrow X \leq 3,\ s(X)$

where q, r, and s are predicates that do not depend on p.
By specializing P w.r.t. $X \geq 3, p(X)$ we would like to derive a new program P_s where the three clauses for p are replaced by the two clauses:

$p_s(X) \leftarrow q(X)$
$p_s(X) \leftarrow X = 3,\ s(X)$.

Indeed, in a context where $X \geq 3$ holds, we have that $X \geq 0$ is true, $X \leq 0$ is false, and $X \leq 3$ is equivalent to $X = 3$.

The technique for specializing CLP programs we present in this paper, follows the approach based on *transformation rules and strategies*. This approach has the advantage of separating the issue of proving the *correctness* of program specialization from that of guaranteeing the *efficiency* of the specialized programs. We consider adaptations of the usual *unfolding* and *folding* rules [3, 19] and variants of these rules [1, 5, 14] for the case of CLP programs. We also consider an extra rule, called *contextual constraint replacement*, which allows us to specialize a set of clauses which define a predicate, say p, with respect to a constraint which is known to hold for the calls of p. Indeed, the example of program specialization given above, can be viewed as an application of our contextual constraint replacement rule (followed by *clause removal* for deleting the second clause).

In Section 3 we present our transformation rules and we give some applicability conditions which ensure that they preserve the least $\mathcal{D}$-model of any CLP program when used in the specified context. These applicability conditions are not implied by those for the various rules considered in [1, 5, 14], and indeed we show that when the folding rule of [1, 5] is combined with our contextual constraint replacement rule, then correctness is not preserved. (The folding rule considered in [14] is always correct, but at the price of strong restrictions.)

In Section 4 we outline a general strategy for deriving very efficient CLP programs by specializing them w.r.t. their context of use. Through an example, we indicate how to direct the use of each transformation rule and, in particular, we address the crucial problem of using the definition rule to introduce new predicate definitions.

Finally, in Section 5 we compare our method for specializing CLP programs to existing methods for partial evaluation of CLP programs [11, 20] based on extensions of Lloyd and Shepherdson's approach [13] (also called *partial deduction* in the literature), and we also compare our method to other specialization techniques based on the unfold/fold transformation rules [2, 7].

2 Preliminaries

The Constraint Logic Programming scheme [8] is an extension of the standard logic programming paradigm which allows the integration of constraints with a predefined interpretation. It defines a class of languages which are parametric in the choice of the constraint domain $\mathcal{D}$, that is, the structure over which computation is performed. Let $\Sigma = (\Pi_c \cup \Pi_u \cup \Phi)$ be a signature defining a set of predicate and function symbols with their associated arity, where

- Π_c is the set of the predicate symbols used for constraints, among which there is the symbol "=", which is interpreted as identity in the domain D (see below);
- Π_u is the set of the symbols for user defined predicates;
- Φ is the set of the function symbols;
- Π_c, Π_u, and Φ are pairwise disjoint sets.

Given a signature Σ, a Σ-structure $\mathcal{D}$ consists of two elements: a non-empty set D and an interpretation which assigns to each n-ary predicate symbol in Π_c a relation over D^n (i.e. a subset of D^n), and to each n-ary function symbol in Φ a function from D^n to D.

A *primitive constraint* has the form $p(t_1, \ldots, t_n)$, where $p \in \Pi_c$ and $t_1, \ldots, t_n$ are terms built from Φ and a denumerable set V of variables. The set of *constraints* contains the constants *true* and *false*, the primitive constraints, and it is closed w.r.t. variable renaming, conjunction, and existential quantification. Constraints will be denoted by lower-case letters $c, d, e, \ldots$. Atomic formulas built using predicate symbols in Π_u, function symbols in Φ, and variables in V, are called *atoms* and will be denoted by upper-case letters $A, B, C, \ldots$. In the following we will use the symbol "," instead of "$\wedge$" to denote the conjunction connective.

A *constraint logic program* is a finite set of *clauses* of the form $A \leftarrow c, B_1, \ldots, B_n$, where $A, B_1, \ldots, B_n$ are atoms and c is a constraint. A *constrained query* is of the form $c, B_1, \ldots, B_n$, where $B_1, \ldots, B_n$ are atoms and c is a constraint; if $n = 1$ it is called *constrained atomic query*.

A $\mathcal{D}$-interpretation I of a formula is an interpretation of the formula over the domain D which agrees with $\mathcal{D}$ on the interpretation of the symbols in $\Pi_c \cup \Phi$, that is, $I \subseteq \mathcal{B}$, where $\mathcal{B} = \{p(d) \,|\, p \text{ is a predicate of arity } n \text{ in } \Pi_u \text{ and } d \in D^n\}$. A $\mathcal{D}$-model $\mathcal{M}$ of a CLP program P is a $\mathcal{D}$-interpretation such that each clause of P is *true* in $\mathcal{M}$. The notion of least Herbrand model for "standard" logic programs is generalized leading to the notion of *least* (w.r.t. set inclusion) *$\mathcal{D}$-model* of a CLP program P, denoted by $lm(P, \mathcal{D})$.

Given a query Q, by $FV(Q)$ we denote the set of *free variables* of Q. Given two atoms with the same predicate symbol $p(t_1, \dots, t_n)$ and $p(u_1, \dots, u_n)$, by $p(t_1, \dots, t_n) = p(u_1, \dots, u_n)$ we mean the conjunction $t_1 = u_1, \dots, t_n = u_n$.

A *valuation* is a function $\sigma : V \to D$. For a term t, by $\sigma(t)$ we denote the element of D obtained by applying the valuation σ to the variables of t. By $\mathcal{D} \models_\sigma c$ we mean that the constraint c, with free variables assigned by σ, is true in $\mathcal{D}$.

3 Transformation rules

We will present here a set of transformation rules for CLP programs. These rules will be applied according to a suitable strategy, described in Section 4, in order to achieve an efficient specialized program which exploits the knowledge of the context where it is used.

Our rules have some differences w.r.t. those studied in previous papers [1, 5, 14], and thus, the correctness results presented in those papers are not applicable in our case. These differences will be discussed as our rules are presented.

The process of deriving a specialized program P_s from an initial program P, is formalized as a sequence of programs $P_0, \dots, P_n$, called a *transformation sequence*, where $P_0 = P$, $P_n = P_s$ and, for $k = 0, \dots, n–1$, P_{k+1} is obtained from P_k by applying one of the following transformation rules: *constrained atomic definition*, *unfolding*, *contextual constraint replacement, constrained atomic folding*, *clause removal*, and *equality introduction and elimination* (see below).

For applying a transformation rule to program P_k of a transformation 1sequence, we will also need some information on the previous steps of construction of the transformation sequence. In particular, we consider two sets associated with $P_0, \dots, P_k$: $Constr_k$ and Def_k, the set of constrained atomic queries and the set of definitions, respectively, introduced up to step k. We take $Constr_0 = \{\}$ and $Def_0 = \{\}$.

3.1 Constrained Atomic Definition

By *constrained atomic definition* (or *definition*, for short), we derive P_{k+1} by adding to P_k a clause δ, called a *definition*, of the form $newp(X_1, \dots, X_n) \leftarrow c, p(t_1, \dots, t_m)$, where (i) *newp* is a predicate symbol not occurring in $P_0, \dots, P_k$; (ii) $X_1, \dots, X_n$ are pairwise distinct variables occurring in $c, p(t_1, \dots, t_m)$; (iii) c is a constraint; (iv) p is a predicate symbol occurring in P_0.

We derive $Def_{k+1} = Def_k \cup \{\delta\}$. We also derive $Constr_{k+1}$ to $Constr_k$ by adding a new constrained atom $d, newp(X_1, \dots, X_n)$ such that $\mathcal{D} \models d \to c$.

The form of definition allowed by our definition rule is less general than the one allowed by [1], where a *conjunction* of atoms may appear in the body of a definition. However, this restricted form of definition is sufficient for many interesting program specializations, while simplifying the problem of controlling the applications of the definition rule (see the strategy presented in Section 4).

Notice that, in contrast to previous approaches, we have the possibility of adding a constrained atom for the newly defined predicate, and this fact will be important for the application of the contextual replacement rule 3.3.

3.2 Unfolding

The unfolding rule allows us to replace a clause δ of P_k with the set of *all* clauses obtained by performing a symbolic computation step w.r.t. an atom A_i in the body of δ and all clauses which define the predicate occurring in A_i in program P_k.

Let δ be a (renamed apart) clause in P_k of the form $A_0 \leftarrow c, A_1, \ldots, A_i, \ldots, A_n$ with $1 \leq i \leq n$, and let $B_{j0} \leftarrow d_j, B_{j1}, \ldots, B_{jn_j}$ for $j = 1, \ldots, m$, be *all* clauses in P_k such that A_i and B_{j0} have the same predicate symbol. For $j = 1, \ldots, m$, let δ_j be the clause

$$A_0 \leftarrow c, d_j, (A_i = B_{j0}), A_1, \ldots, A_{i-1}, B_{j1}, \ldots, B_{jn_j}, A_{i+1}, \ldots, A_n$$

Then, by *unfolding δ in P_k w.r.t. the atom A_i*, we derive the new program $P_{k+1} = (P_k - \{\delta\}) \cup \{\delta_j \mid j = 1, \ldots, m\}$.

Notice that, unlike the unfolding rule in [5, 14], the satisfiability of the constraint $c, d_j, (A_i = B_{j0})$ is not checked during the application of the unfolding rule, as satisfiability checks can be performed by the clause removal rule presented in 3.5. Our presentation allows for a better separation between the treatments of the logic and the constraint parts of the programs to be transformed.

3.3 Contextual Constraint Replacement

Let δ be a (renamed apart) clause in P_k of the form $p(t) \leftarrow c, A_1, \ldots, A_m$, where t is a vector of terms. Let *p-Calls* be the set of constrained atoms $d, p(u)$ such that $B_0 \leftarrow d, B_1, \ldots, p(u), \ldots, B_n \in P_k$, where u is a vector of terms. Suppose that for all constrained atoms $d, p(u) \in p\text{-}Calls \cup Constr_k$, we have that:

$$\mathcal{D} \models \exists X_1, \ldots, X_i (d, (u = t), c) \leftrightarrow \exists Y_1, \ldots, Y_j (d, (u = v), c') \quad (1)$$

where $\{X_1, \ldots, X_i\} = FV(t, c) - FV(A_1, \ldots, A_m)$, $\{Y_1, \ldots, Y_j\} = FV(v, c') - FV(A_1, \ldots, A_m)$, v is a vector of terms, and c' is a new constraint. Then we derive P_{k+1} from P_k by replacing δ with the clause $p(v) \leftarrow c', A_1, \ldots, A_m$.

Notice that the following *constraint replacement rule* [1, 14] can be viewed as a particular case of our contextual constraint replacement rule, because condition (2) below implies our condition (1) for the applicability of the contextual replacement rule. Let δ be a clause in P_k of the form $p(t) \leftarrow c, A_1, \ldots, A_m$, suppose that

$$\mathcal{D} \models \exists X_1, \ldots, X_i c \leftrightarrow \exists Y_1, \ldots, Y_j c' \quad (2)$$

where $\{X_1, \ldots, X_i\} = FV(c) - FV(t, A_1, \ldots, A_m)$ and $\{Y_1, \ldots, Y_j\} = FV(c') - FV(t, A_1, \ldots, A_m)$. Then we derive P_{k+1} from P_k by replacing δ with the clause $p(t) \leftarrow c', A_1, \ldots, A_m$.

As already mentioned, the example presented in the Introduction is an application of our contextual constraint replacement rule, where we assume that $X \geq 3, p(X)$ is the only constrained atom which belongs to the $p\text{-}Calls \cup Constr_k$ set.

The constraint replacement rule of [5] is more general than the ones considered in [1, 14], but it is based on a condition on successful derivations (which is undecidable) and it does not allow us to deal with the example presented in the Introduction (thus its power is incomparable with our contextual replacement rule).

To see an example where the condition on the *p-Calls* set is necessary for correctness, let us consider the following program P_k:

$p(X) \leftarrow X=0,\ p(1)$
$p(Y) \leftarrow Y=1$

and assume that $Constr_k = \{(Y=0, p(Y))\}$. Condition (1) disallows the derivation of P_{k+1} by the replacement of the second clause of P_k with $p(Y) \leftarrow Y=0,\ Y=1$, because, for the constrained atom $(X=0, p(1)) \in$ *p-Calls*, $\mathcal{D} \not\models \exists X, Y(X=0, (1=Y), (Y=1)) \leftrightarrow \exists X, Y(X=0, (1=Y), (Y=0, Y=1))$. Notice that this derivation is incorrect, because $p(0) \in lm(P_k, \mathcal{R})$ while $p(0) \notin lm(P_{k+1}, \mathcal{R})$.

3.4 Constrained Atomic Folding

Let γ be a (variant of a) clause in P_k of the form

$$A_0 \leftarrow c, d, A_1, \ldots, A_{i-1}, p(t), A_{i+1}, \ldots, A_m$$

where $1 \leq i \leq m$ and t is a vector of terms, and let δ be a clause in Def_k of the form $newp(X) \leftarrow d, p(t)$, where X is a vector of variables. By *folding* γ *by* δ w.r.t. $d, p(t)$, we derive the new clause γ' of the form

$$A_0 \leftarrow c, A_1, \ldots, A_{i-1}, newp(X), A_{i+1}, \ldots, A_m$$

provided that the following conditions hold:

1. for each variable $Y \in vars(\delta) - \{X\}$ (that is, Y is an *internal* variable of δ) Y does not occur in $A_0, c, A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m$;

2. the predicate symbol occurring in A_0 is an *old* one, that is, it has not been introduced by the definition rule, or γ is the result of at least one unfold transformation;

3. if by an application of the contextual replacement rule at step $h < k$, a clause of the form $newp(v) \leftarrow e, B_1, \ldots, B_n$ has been replaced by $newp(w) \leftarrow e', B_1, \ldots, B_n$, then $\mathcal{D} \models \exists V_1, \ldots, V_r(c, X = v, e) \leftrightarrow \exists W_1, \ldots, W_s(c, X = w, e')$ where $\{V_1, \ldots, V_r\} = FV(v, e) - FV(B_1, \ldots, B_n)$ and $\{W_1, \ldots, W_s\} = FV(w, e') - FV(B_1, \ldots, B_n)$.

¿From P_k we derive $P_{k+1} = (P_k - \{\gamma\}) \cup \{\gamma'\}$.

Our folding rule differs from the folding rules considered in [1, 5, 14] for the reasons mentioned below.

- By using our folding rule, a clause can only be folded by a definition which has *at most one* atom in the body. In this respect our rule is more restrictive than the rules in [1, 5, 14], which allow several atoms to occur in the body of a definition, but our choice is adequate in the framework of program specialization (and indeed our framework extends the usual one of *partial deduction* [13]).

- In contrast to [1, 5], our folding rule allows us to eliminate occurrences of constraints: the constraint d in the body of clause γ does not occur in the body of clause γ'. This fact is important when we want to improve efficiency by means of the program specialization strategy presented in Section 4.

- Similarly to [1, 5], and unlike [14], our folding rule is not necessarily *reversible*, that is, we may derive recursive clauses by folding with previously introduced definitions, which do not necessarily belong to the current program. Also this fact is important for our program specialization strategy, where the folding rule is used for exploiting *at each recursive call* the efficiency improvements achieved by unfolding and constraint replacement. Our condition 2 is similar to the conditions of [1, 5], which ensure that the introduction of new recursive clauses preserves the semantics of interest.

- Finally, condition 3 is a very relevant difference w.r.t. all previous approaches, because it is needed for ensuring the correctness of the folding rule when used together with the contextual replacement rule. To see why this extra condition is needed, let us consider the following CLP($\mathcal{R}$) program P_0:
  $p(X) \leftarrow X\!=\!0,\ p(1)$
  $p(X) \leftarrow X\!=\!1$
  Let us introduce, by the definition rule, the clause $newp(X) \leftarrow p(X)$ and the constrained query $X = 0, newp(X)$. Thus, we have derived $P_1 = P_0 \cup \{newp(X) \leftarrow p(X)\}$ and $Constr_1 = \{(X\!=\!0, newp(X))\}$.
  We now unfold the clause for *newp* and we derive:
  $P_2 = P_0 \cup \{newp(X) \leftarrow X\!=\!0, p(1), \quad newp(X) \leftarrow X\!=\!1\}$
  By contextual constraint replacement we derive:
  $P_3 = P_0 \cup \{newp(X) \leftarrow X\!=\!0, p(1), \quad newp(X) \leftarrow X\!=\!0, X\!=\!1\}$
  If we now fold without fulfilling condition 3, then we get:
  $P_4 = P_0 \cup \{newp(X) \leftarrow X\!=\!0, newp(1), \quad newp(X) \leftarrow X\!=\!0, X\!=\!1\}$
  and this transformation step is incorrect, because $newp(0) \in lm(P_1, \mathcal{R})$ while $newp(0) \notin lm(P_4, \mathcal{R})$.

3.5 Clause Removal

Let δ be a clause in P_k of the form $A_0 \leftarrow c, A_1, \ldots, A_n$. If $\mathcal{D} \models \neg\exists X_1, \ldots, X_i c$, where $X_1, \ldots, X_i$ are the free variables of c, then we derive $P_{k+1} = P_k - \{\delta\}$. We say that c is an *inconsistent* constraint.

The clause removal rule is also present in [1, 5, 14].

3.6 Equality Introduction and Elimination

Let γ be a clause of the form $A_0 \leftarrow c, A_1, \ldots, A_n$, let δ be a clause of the form $B_0 \leftarrow d, (X = t), B_1, \ldots, B_n$ such that X does not occur in t and $(A_0 \leftarrow c, A_1, \ldots, A_n) = (B_0 \leftarrow d, B_1, \ldots, B_n)\{X/t\}$. If $\gamma \in P_k$, then, by *equality introduction*, we derive $P_{k+1} = (P_k - \{\gamma\}) \cup \{\delta\}$. If $\delta \in P_k$, then, by *equality elimination,* we derive $P_{k+1} = (P_k - \{\delta\}) \cup \{\gamma\}$.

3.7 Correctness of the Transformation Rules

We now present our main theoretical result, which ensures the correctness of program specialization when it is performed by means of the transformation rules presented in this section.

Theorem (Correctness of Program Specialization). Let us consider a transformation sequence $P_0, \ldots, P_n$ and the associated sets Def_n and $Constr_n$. Let us also consider $newp(X_1, \ldots, X_h) \leftarrow c, p(t_1, \ldots, t_m) \in Def_n$ and $(d, newp(X_1, \ldots, X_h)) \in Constr_n$ with $\mathcal{D} \models d \rightarrow c$.
Then, for every valuation σ such that $\mathcal{D} \models_\sigma d$ we have:
$p(\sigma(t_1), \ldots, \sigma(t_m)) \in lm(P_0, \mathcal{D})$ iff $newp(\sigma(X_1), \ldots, \sigma(X_h)) \in lm(P_n, \mathcal{D})$.

For lack of space, we do not present here the proof of our correctness theorem. However, we would like to notice that our this theorem is not a consequence of the results proved in [1, 5, 14] because, as already mentioned, our rules are not particular cases of the rules considered in those papers (see, in particular the contextual constraint replacement and folding rules).

4 A Strategy for Contextual Specialization of CLP Programs

We now describe a general strategy for guiding the construction of a transformation sequence towards the goal of deriving efficient specialized CLP programs. Our strategy is a generalization of the strategies for the partial evaluation of standard logic programs using unfold/fold transformation rules (see for instance the strategies implemented by the partial evaluators Mixtus [18] and Paddy [17]).

Contextual Specialization Strategy
Input: A CLP program P and a constrained atomic query $c, p(t_1, \ldots, t_m)$.
Output: A CLP program P_s and a constrained atomic query $c, p_s(X_1, \ldots, X_h)$ such that, for every valuation σ, if $\mathcal{D} \models_\sigma c$, then

$$p(\sigma(t_1), \ldots, \sigma(t_m)) \in lm(P_0, \mathcal{D}) \quad \text{iff} \quad p_s(\sigma(X_1), \ldots, \sigma(X_h)) \in lm(P_s, \mathcal{D}).$$

Initialization
$NewConstr := \{(c, p_s(X_1, \ldots, X_h))\}$;
$NewDef := \{p_s(X_1, \ldots, X_h) \leftarrow c, p(t_1, \ldots, t_m)\}$
where $\{X_1, \ldots, X_h\} \subseteq FV(c, t_1, \ldots, t_m)$;
$Constr := NewConstr$; $Def := NewDef$;

while ($NewDef \neq \{\}$) **do**

1. *Unfold* each clause in *NewDef* w.r.t. the leftmost atom in its body. Apply the *clause removal* rule and remove each clause derived by unfolding which has an inconsistent constraint in its body. Let Γ be the resulting set of clauses.

2. **while** there is a clause γ in Γ such that by applying the unfolding rule w.r.t an atom A in the body of γ, we get *at most one* clause with a satisfiable constraint in its body
   do
   unfold γ w.r.t. A and then possibly apply the *clause removal* rule to the resulting clauses, thereby obtaining a set Γ_u of clauses.
   $\Gamma := (\Gamma - \{\gamma\}) \cup \Gamma_u$.
   end

3. Apply the *constraint replacement* rule and the *equality introduction/elimination* rule to the clauses in Γ so to allow for applications of the folding rule (see next step);
   if for each atom B occurring in the body of a clause γ in Γ there exists a constraint d in the body of γ and a clause $q_s(Y_1, \ldots Y_k) \leftarrow d, B$ in *Def* by which we can fold γ w.r.t. d, B (see Section 3.4 for the applicability conditions of the folding rule)
   then *NewDef* $:= \{\}$ and *NewConstr* $:= \{\}$
   else by applications of the *definition* rule, introduce a new set of clauses *NewDef* such that for each atom B occurring in the body of a clause γ in Γ there exists a constraint d in the body of γ and a clause $q_s(Y_1, \ldots Y_k) \leftarrow d, B$ in *Def* $\cup$ *NewDef* by which we can fold γ w.r.t. d, B. For each clause in *NewDef* of the form $q_s(Y_1, \ldots, Y_k) \leftarrow d, B$ add the constrained atomic query $(d, q_s(Y_1, \ldots, Y_k))$ to *NewConstr*.
   endif
   Def $:=$ *Def* $\cup$ *NewDef*; *Constr* $:=$ *Constr* $\cup$ *NewConstr*;

4. For each atom B occurring in the body of a clause γ in Γ consider a constraint d in the body of γ, *fold* γ w.r.t. d, B by using a clause in *Def*;

5. Apply the *contextual constraint replacement* rule using constrained queries from *Constr*. This step is performed to simplify the constraints occurring in the bodies of the clauses defining a predicate, by taking advantage of the information provided by the contexts in which the predicate is called. Indeed, if we know that a certain constraint holds at each call of a predicate, by applying the contextual constraint replacement rule we may avoid to repeatedly check the satisfiability of that constraint at each recursive call;

end

For performing unfolding steps in our Contextual Specialization Strategy we have adapted the unfolding strategy called *determinate unfolding* [6]. This unfolding strategy does not terminate in general. Further termination problems may arise because our strategy may introduce infinitely many new definitions. To ensure termination we may need a suitable *generalization* strategy for limiting the number of definitions to be introduced (see, for instance, [15]). However the detailed study of termination and generalization issues is beyond the scope of the present paper where we want to analyze the use of contextual information for program specialization.

Now we present an example of the application of our transformation rules and strategy to the following CLP($\mathcal{R}$) program:

$$\begin{array}{rcll}
sm(A,B) & \leftarrow & C\!=\!D,\ E\!=\!F,\ length(A,C),\ sum(A,E), & \\
 & & prefix(G,B),\ length(G,D),\ sum(G,F). & (1)\\
sm(A,[B|C]) & \leftarrow & B\!\geq\!0,\ sm(A,C). & (2)\\
sum([\,],A) & \leftarrow & A\!=\!0. & (3)\\
sum([A|B],C) & \leftarrow & A\!\geq\!0,\ C\!\geq\!0,\ C\!=\!A\!+\!D,\ sum(B,D). & (4)\\
length([\,],A) & \leftarrow & A\!=\!0. & (5)\\
length([A|B],C) & \leftarrow & A\!\geq\!0,\ C>0,\ C\!=\!D\!+\!1,\ length(B,D). & (6)\\
prefix([\,],A). & & & (7)\\
prefix([A|B],[C|D]) & \leftarrow & A\!=\!C,\ prefix(B,D). & (8)
\end{array}$$

The declarative reading of such a program is: "a pattern A of length L, such that the sum of its elements is S, "matches" a string B if B contains a substring G of length L and the sum of the elements of G is S".

We want to derive a specialized program w.r.t. the constrained query

$$A\!+\!B\!+\!C\!\leq\!5,\ A\!\geq\!0,\ B\!\geq\!0,\ C\!\geq\!0,\ sm([A,B,C],D)$$

so we introduce by the definition rule 3.1 the following clause:

$$sm_s(A,B,C,D)\ \leftarrow\ A\!+\!B\!+\!C\!\leq\!5,\ A\!\geq\!0,\ B\!\geq\!0,\ C\!\geq\!0,\ sm([A,B,C],D). \quad (9)$$

and we get $Def_1 = \{9\}$ and $Constr_1 = \{((A+B+C \leq 5,\ A \geq 0,\ B \geq 0,\ C \geq 0),\ sm_s(A,B,C,D))\}$. Next, we apply the unfolding rule 3.2 to clause 9 followed by an application of the equality elimination rule 3.6 obtaining:

$$\begin{array}{rcll}
sm_s(A,B,C,D) & \leftarrow & A\!+\!B\!+\!C\!\leq\!5,\ A\!\geq\!0,\ B\!\geq\!0,\ C\!\geq\!0,\ E\!=\!F,\ G\!=\!H, & \\
 & & length([A,B,C],E),\ sum([A,B,C],G), & \\
 & & prefix(I,D),\ length(I,F),\ sum(I,H). & (10)\\
sm_s(A,B,C,[D|E]) & \leftarrow & A\!+\!B\!+\!C\!\leq\!5,\ A\!\geq\!0,\ B\!\geq\!0,\ C\!\geq\!0, & \\
 & & D\!\geq\!0,\ sm([A,B,C],E). & (11)
\end{array}$$

For the sake of readability, in the following we assume to implicitly use the equality introduction and elimination rule 3.6 when we deal with equalities between terms. By unfolding clause 10 we get:

$$\begin{array}{rcll}
sm_s(A,B,C,D) & \leftarrow & A\!+\!B\!+\!C\!\leq\!5,\ A\!\geq\!0,\ B\!\geq\!0,\ C\!\geq\!0, & \\
 & & E\!=\!F,\ G\!=\!H,\ A\!\geq\!0,\ E>0,\ E\!=\!I\!+\!1, & \\
 & & length([B,C],I),\ sum([A,B,C],G),\ prefix(J,D), & \\
 & & length(J,F),\ sum(J,H). & (12)
\end{array}$$

along with

$$\begin{array}{rcll}
sm_s(A,B,C,D) & \leftarrow & A\!+\!B\!+\!C\!\leq\!5,\ A\!\geq\!0,\ B\!\geq\!0,\ C\!\geq\!0,\ E\!=\!F,\ G\!=\!H, & \\
 & & E\!=\!0,\ [\,]\!=\![A,B,C],\ sum([A,B,C],G), & \\
 & & prefix(J,D),\ length(J,F),\ sum(J,H). & (13)
\end{array}$$

Clause 13 is removed by an application of the clause removal rule 3.5. Now we proceed by unfolding the first occurrence of the predicate *length* in the body of clause 12 and we get

$$\begin{array}{lll} sm_s(A,B,C,D) & \leftarrow & A+B+C\leq 5,\ A\geq 0,\ B\geq 0,\ C\geq 0,\ E=F,\ G=H,\ A\geq 0, \\ & & E>0,\ E=I+1,\ B\geq 0,\ I>0,\ I=J+1,\ C\geq 0,\ J>0, \\ & & J=K+1,\ K=0,\ sum([A,B,C],G),\ prefix(L,D), \\ & & length(L,F),\ sum(L,H). \end{array} \tag{14}$$

After an application of the constraint replacement rule 3.3 we have

$$\begin{array}{lll} sm_s(A,B,C,D) & \leftarrow & A+B+C\leq 5,\ A\geq 0,\ B\geq 0,\ C\geq 0, \\ & & 3=F,\ G=H,\ sum([A,B,C],G), \\ & & prefix(J,D),\ length(J,F),\ sum(J,H). \end{array} \tag{15}$$

(recall that our constraint replacement rule 3.3 may also be applied to simplify a constraint within a clause). By applying the determinate unfolding strategy we get

$$\begin{array}{lll} sm_s(A,B,C,[D,E,F|G]) & \leftarrow & A+B+C\leq 5,\ A\geq 0,\ B\geq 0,\ C\geq 0, \\ & & D+E+F=A+B+C, \\ & & D\geq 0,\ E\geq 0,\ F\geq 0. \end{array} \tag{16}$$

and, after an application of the contextual constraint replacement rule to clause 16

$$sm_s(A,B,C,[D,E,F|G]) \leftarrow D+E+F=A+B+C,\ D\geq 0,\ E\geq 0,\ F\geq 0. \tag{17}$$

Now we fold clause 11 by clause 9 and we obtain the following recursive clause

$$sm_s(A,B,C,[D|E]) \leftarrow D\geq 0,\ sm_s(A,B,C,E). \tag{18}$$

It should be noticed here that the specialized program we have obtained, made out of clauses 17 and 18, when used for the constrained query

$$A+B+C\leq 5,\ A\geq 0,\ B\geq 0,\ C\geq 0,\ sm_s(A,B,C,D)$$

exploits the fact that the pattern $[A,B,C]$ has length 3 and the sum of its elements is $A+B+C$ and actually avoids to rediscover this knowledge each time backtracking occurs, as the original program does.

Another source of performance improvement may come from the detection and the removal of redundant arguments. Indeed, it is often the case that programs which have been generated by automatic strategies, like the one we have proposed at the beginning of this section, contain predicate definitions using arguments which are not needed. In the literature some works have addressed the issue of removing unnecessary structures or variables from logic programs (see, for instance, [6] and [12]). In our setting, however, the problem is slightly different. Sometimes it might be possible to perform an equivalent computation if we replace redundant arguments within a predicate with a different set of arguments carrying only the information actually used in the computation.

For example, by an analysis of the specialized program obtained so far, one can observe that the arguments A, B, C do not change for recursive calls to sm_s. A more efficient program can be obtained if we consider a different definition of the specialized predicate, which does not depend on A, B, C. Indeed, suppose we introduce by definition

$$sm'_s(R, D) \leftarrow A{+}B{+}C{=}R,\ R{\leq}5,\ A{\geq}0,\ B{\geq}0,\ C{\geq}0,\ sm([A, B, C], D). \quad (19)$$

and we get $Def_1 = \{19\}$ and $Constr_1 = \{((A{+}B{+}C{=}R,\ R{\leq}5,\ A{\geq}0,\ B{\geq}0,\ C{\geq}0),\ sm'_s(R, D))\}$. Now, by applying our strategy, we end with the following two clauses defining sm'_s:

$$\begin{aligned} sm'_s(R, [D, E, F|G]) &\leftarrow D{+}E{+}F{=}R,\ D{\geq}0,\ E{\geq}0,\ F{\geq}0. && (20)\\ sm'_s(R, [D|E]) &\leftarrow D{\geq}0,\ sm'_s(R, E). && (21)\end{aligned}$$

This second version of the program, while keeping the advantages achieved by the first version, avoids to carry the invariant arguments A, B, C and to recompute the expression $A{+}B{+}C$. This optimization is very relevant because our derivation can be generalized to the case where the given pattern is very long and/or the computation over it is a very expensive task. In this case the optimized version of the specialized program may introduce a significant efficiency improvement. This is to show how crucial is the choice of the predicates introduced by definition.

5 Related Work and Conclusions

We have presented some transformation rules and strategies for the specialization of constraint logic programs to the context where they are to be used. Our method extends related methods for the *partial evaluation* of logic programs because the context of use is specified by means of a constraint which may be satisfied by a possibly infinite set of values, instead of a single value.

Indeed, our specialization method belongs to the family of methods for the specialization of program w.r.t. *properties* of the context. Among these, we would like to mention: (1) the methods for specializing logic programs presented in [2] and [16] and (2) the method for *parametrized partial evaluation* of functional programs presented in [4].

The works [2] and [16] present very general approaches which, however, do not seem to be amenable to full mechanization. In particular, in [2] most of the basic transformation rules cannot be easily automated because their applicability conditions depend on undecidable properties of the programs to be transformed. In this paper we consider CLP programs, instead of logic programs, and we describe the properties of the context by means of constraints, instead of predicates defined by logic programs. By doing so, we are able to consider much more effective transformation rules, whose applicability conditions are decidable, if we assume that the *constraint theory is decidable* (as is often the case).

The work in [4] is concerned with a strict first order functional programming language and, most importantly, the main means for specifying a property of the context w.r.t. which a program has to be specialized, is based on *abstract interpretation.*

Our transformation rules are intended for use in program specialization techniques and, in this respect, they are more restrictive than the transformation rules for CLP programs considered in [1, 5, 14]. However, as we have extensively discussed in Section 3, our transformation rules are *not* particular cases of those considered in [1, 5, 14]. In particular, our *contextual constraint replacement rule* allows us to replace a constraint occurring in the body of a clause by another constraint which is equivalent in a given context. For instance, consider the program P_0:

$p(X) \leftarrow X \geq 0,\ q(X)$

where q is a predicate that does not depend on p. By contextual constraint replacement w.r.t. the constrained atom $X \geq 3, p(X)$, we may derive P_1:

$p(X) \leftarrow q(X)$

($X \geq 0$ is true in the context where $X \geq 3$ holds). Notice that P_1 cannot be obtained by unfolding/folding, because it has a different least $\mathcal{D}$-model w.r.t. P_0.

We have pointed out that the correctness results for the transformation rules presented in [1, 5, 14] do not carry over our framework, because we have modified the folding rule so as to be correct when used in combination with contextual constraint replacement.

We have also outlined a general strategy for specializing CLP programs by using our rules. This strategy extends various techniques based on the unfolding and folding rules for partially evaluating logic and constraint logic programs (see, for instance, [7, 17, 18]). Our folding rule is more powerful than the ones considered in those papers and, besides that, we have shown in the matching example that our contextual constraint replacement rule may play an important role in improving the efficiency of the specialized program.

Finally, the papers [11, 20] report on some correctness results for *partial deduction*, that is, partial evaluation of logic programs based on Lloyd and Shepherdson's approach [13]. Besides various technical issues that make partial deduction and unfolding/folding-based program specialization different, we would like to point out that the frameworks described in [11, 20] do not allow for contextual constraint replacement. Indeed, from the atomic query $X \geq 3, p(X)$ and program P_0 above we derive the *resultant*:

$p(X) \leftarrow X \geq 3,\ q(X)$

where the constraint $X \geq 3$ has to be solved at evaluation time, even if we know that it holds at specialization time. In contrast to partial deduction, our approach is able to fully exploit the statically available information on the context of use of a program.

Acknowledgements

We thank Alberto Pettorossi for stimulating discussions and helpful suggestions on the subject of this paper. The program derivations where supported by the interactive MAP transformation system implemented by Sophie Renault.

This work has been partially supported by Progetto Coordinato del CNR 'Verifica, Analisi e Trasformazione di Programmi Logici' and Progetto Cofinanziato MURST 'Tecniche Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software'.

References

[1] N. Bensaou and I. Guessarian. Transforming constraint logic programs. *Theoretical Computer Science*, (206):81–125, 1998.

[2] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.

[3] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[4] C. Consel and S.C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[5] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.

[6] J. P. Gallagher. Tutorial on specialization of logic programs. In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.

[7] T. J. Hickey and D. A. Smith. Towards the partial evaluation of CLP languages. In *Proceedings ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '91, New Haven, CT, USA*, SIGPLAN Notices, 26, 9, pages 43–51. ACM Press, 1991.

[8] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[9] J. Jaffar, M. Maher K. Marriott, and P. Stuckey. The semantics of constraint logic programming. *Journal of Logic Programming*, 37:1–46, 1998.

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

[11] M. Leuschel and D. De Schreye. Constrained partial deduction. In B. Freitag F. Bry and D. Seipel, editors, *Proceedings of the 12th Workshop Logische Programmierung (WLP'97)*, pages 116–126, Munich, Germany, September 1997.

[12] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation, Proceedings LoPSTr '96, Stockholm, Sweden*, Lecture Notes in Computer Science 1207, pages 83–103. Springer-Verlag, 1996.

[13] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[14] M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.

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

[16] A. Pettorossi and M. Proietti. A theory of logic program specialization and generalization for dealing with input data properties. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, Lecture Notes in Computer Science 1110, pages 386–408. Springer-Verlag, 1996.

[17] S. Prestwich. Online partial deduction of large programs. In *Proceedings ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 111–118. ACM Press, 1993.

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

[19] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.

[20] A. Wrzos-Kaminska. Partial evaluation in constraint logic programming. In Z.W. Ras and M. Michalewicz, editors, *Proceedings of the 9th International Symposium on Foundations of Intelligent Systems*, Lecture Notes in Computer Science 1079, pages 98–107, Zakopane, Poland, 1996. Springer-Verlag.